UniCryptJS

Marcel Portillo

Biel, Spring 2016

Index

1 Einleitung

1.1 Aufgabenstellung

UniVote ist eine von der Berner Fachhochschule entwickelte online Abstimmungsplattform [2]. Für die kryptographischen Operationen wird die UniCrypt Bibliothek verwendet [24]. Da nun aber auch frontendseitig verschlüsselt werden können muss, braucht es eine Entsprechung in JavaScript. Bisher war diese direkt in UniVote eingebunden. Sinnvoller wäre eine eigenständige JavaScirpt Bibliothek, die eine möglichst ähnliche Funktionsweise wie die Java Version aufweist. Ziel der Bachelorarbeit ist es ein Konzept zu entwickeln, durch das eine Portierung ermöglicht wird. Ein Benutzer der UniCryptJS Bibliothek sollte sich auf Anhieb zurechtfinden können, wenn er bereits mit UniCrypt gearbeitet hat.

1.2 Vorgehensweise

1.2.1 Liste der Konzepte

In einem ersten Schritt gilt es eine Übersicht über die in UniCrypt verwendeten Konzepte aus Java zu gewinnen. Grundsätzlich gibt es Unterschiede zwischen einer kompilierten und einer Interpreter basierten Sprache. Dazu kommt, das JavaScript dynamisch typisiert ist und viele Typenkonversionen automatisch ausführt.

UniCrypt macht beispielsweise exzessiven Gebrauch von Abstrakten Klassen, Interfaces, dem Überladen von Methoden oder der Typisierung. Eine JavaScript Implementierung der Bibliothek muss diese Konzepte ebenfalls unterstützen, damit die gewünschte Ähnlichkeit bei der Verwendung gewährleistet werden kann.

1.2.2 Verständnis von Java und JavaScript erarbeiten

Während der Ausbildung an der Fachhochschule wurde mehrheitlich Java gelehrt. Während dem Programmieren verwendete ich intuitiv die Sprachkonstrukte. Sie sind einfach da und gehören zu den Paradigmen der objektorientierten Programmierung mit einer kompilierten Sprache. Viele dieser Konzepte kennt JavaScript allerdings nicht. Um in der Interpretersprache diese simulieren zu können, braucht es eine ganz andere Auseinandersetzung mit dem Thema. Folgende Überlegungen werden notwendig:

- Was ist der Unterschied zwischen Interface und abstrakter Klasse?
- Ist eine Interface nicht eine rein abstrakte Klasse [6]?
- Verhält sich eine abstrakte Klasse bei der Vererbung gleich wie eine herkömmliche?

Ein tieferes Verständnis der Funktionsweise von Java Sprachkonstrukten ist erforderlich. Entscheidend ist das Nachbilden phänotypische Verhalten und nicht die technische Implementierung im Compiler von Java.

Ich hatte bereits vor der Bachelorarbeit mit JavaScript gearbeitet. Trotzdem fehlte das Verständnis für die Eigenheiten einer prototypbasierten Sprache. Für jemanden aus der Java Welt erscheinen Konstrukte wie Closures und die "prototype chain" nicht intuitiv logisch. Objekte und Funktionen verhalten sich nicht erwartungsgemäss. Eine vertiefte Einarbeitung in JavaScript ist vonnöten.

1.2.3 Design Entscheidungen

Aufgrund der Eigenheiten von JavaScript und der Liste aus benötigten Java-Sprachkonstrukten müssen Entscheidungen getroffen werden. Nicht für alle muss eine Entsprechung gefunden werden. In manchen Fällen ist eine Implementierung in JavaScript nicht sinnvoll und es ist besser auf Konventionen zurückzugreifen.

1.2.4 Framework entwickeln

Viele Vorgänge bei der Portierung sind repetitiv. Um Redundanzen im Code zu vermeiden, lohnt es sich wiederkehrende Vorgänge durch ein Framework zu abstrahieren. Der Funktionsumfang des Frameworks muss festgelegt werden.

1.2.5 Portierung und Testing

Der eigentliche Programmcode von UniCrypt muss übernommen werden. Mithilfe des Frameworks werden die Klassen in JavaScript abgebildet. Die Methoden werden an die Sprachsyntax angeglichen. Das Framework simuliert die Java-Sprachkonstrukte. Unittests werden geschrieben um das Framework und die UniCryptJS Klassen zu testen.

2 Projektplanung

2.1 Ursprüngliche Planung

2.1.1 22.03.2016 bis 16.03.2016

Es gilt eine Liste über die von UniCrypt verwendeten Sprachkonstrukte aus Java zu erstellen. Welche Konzepte müssen in JavaScript abgebildet werden, um eine ähnliche Verwendung der UniCryptJS zur Verfügung zu stellen wie in bei der Java Bibliothek? Erstellung eines Semesterplans. Auseinandersetzung mit der Funktionsweise von JavaScript und Dokumentation dessen.

2.1.2 16.03.2016 bis 27.03.2016

Es gilt ein Konzept für Javascript auszuarbeiten. Einen Bericht erstellen über die Funktionalität von Javascript und die Abgrenzung zu Java. Entwicklung eines Konzept zur Abbildung aller Konzepte von Java die vonnöten sind. Sich überlegen, ob die Verwendung eines Frameworks sinnvoll erscheint. Mithilfe dieses Konzeptes und dieser Überlegungen kann entschieden werden, welches Vorgehen für die Portierung angewendet wird.

Alle Konzepte die in der Liste an Funktionalitäten vom 16.03.20216 aufgeschrieben wurden, müssen entweder in diesem Portierungskonzept aufgenommen werden (mit einem Vorschlag zur Implementierung) oder aber mit Begründung verworfen werden (wegen zu hohem Implementationsaufwand oder da rein durch die Spezifikation von JavaScript bedingt nicht umsetzbar sind).

2.1.3 27.03.2016 bis 17.04.2016

UniCrypt enthält einen Teilbereich, der die mathematischen Konzepte abbildet. Darin enthalten sind unter anderem die zyklischen modularen Gruppen. Eine dieser Klassen muss lauffähig umgesetzt worden sein. Dazu muss ein Wrapper erstellt werden, der eine thirdparty BigInteger Implementierung umfasst. So kann gewährleistet werden, dass bei einer Änderung des verwendeten Scripts UniCryptJS nicht umgeschrieben werden muss, da nur der Wrapper angepasst werden muss. Alle Konzepte sind vorhanden (entweder im Framework oder per Konvention). Dazu gehören beispielsweise die Prinzipien der immutable objects, Vererbung, Interfaces, Methodenüberladung, abstrakte Klassen und statische Methoden. Dies mit vollständigen Unittests für das Framework und für diese eine Klasse. Diese sind in Übereinstimmung mit den Tests der Java Bilbiothek.

2.1.4 17.04.2016 bis 01.05.2016

Konzept zur Kommunikation zwischen UniCryptJS und UniCrypt. Erste Tests für die exakte Übermittlung übers Netz. Diese muss aufs Bit genau stimmen. Ansonsten ist eine Entschlüsslung nicht mehr möglich. Ebenfalls sollte diese Übertragung mit Unittests automatisiert testbar sein. Weitere benötigte Klassen für eine Verschlüsselung sollten portiert werden.

2.1.5 01.05.2016 bis 15.05.2016

Kommunikation zwischen Javascript und Java Version ist funktionstüchtig. Dies in Form eines Proof-Of-Concept exemplarisch für die bis dahin vorhandenen Klassen aus den Mathematik Teil der Bibliothek. Je nach Stand der Portierung werden nur testweise Daten hin und her gesandt. Im Idealfall handelt es sich bereits um eine komplexere Funktionalität wie ein Pederson Commitment oder eine ElGamal Verschlüsselung.

2.1.6 15.05.2016 bis 29.06.2016

Portierung möglichst vieler Klassen und Automatisierung der Kommunikation über das Netzwerk.

2.1.7 29.06.2016 bis 06.06.2016

Unicrypt Portierung sollte abgeschlossen sein für die spezifizierten Klassen GStarMod, ZStarMod und ZMod und zwar mit allen Abhängigkeiten. Eine ElGamal Verschlüsselung ist lauffähig und ein Pederson Commitment kann erstellt werden. Der Code wurde vollständig dokumentiert. Software sollte fertig sein zur Abgabe. Alle Designentscheidungen sind klar ersichtlich gekennzeichnet und im Bericht beschrieben.

2.1.8 06.06.2016 bis 17.06.2016

Finaltag und Präsentation der Arbeit. Software und Bericht sind fertig und wurden abgegeben. Fehlende Teile werden ergänzt und ein kleiner webbasierter Prototyp wird zu Demonstrationszwecken hergestellt.

2.2 Umsetzung

2.2.1 Probleme

Ich habe mich bereits früh in der Bachelorarbeit für die Entwicklung eines Frameworks entschieden. Es schien mir der beste Ansatz zu sein, um die Portierung vornehmen zu können. Ich ging mit der Annahme an die Arbeit, dass sobald das Framework abgeschlossen ist, der Java Code eins zu eins übernommen werden kann. Ich verwendete Zeit zur Erstellung des Frameworks, musste aber bald merken, dass dies ein zu ambitioniertes Vorhaben für die kurze Zeit der Bachelorarbeit ist. Einige Konzepte aus der Java Welt liessen sich rasch implementieren. Bei genauerer Betrachtung, kamen aber mehr Anforderungen laufend dazu. Solange nicht ein gewisser Stand beim Framework erreicht war, konnte ich noch nicht mit der UniCrypt Portierung beginnen.

Es war das Ziel bis zum 17.04.16 eine erste Klasse aus dem Mathematikteil der UniCrypt Bibliothek lauffähig portiert zu haben. Allerdings war dies zu dem damaligen Zeitpunkt nicht möglich gewesen, da das Framework immer noch nicht die notwendigen Konzepte unterstütze.

Erst spät wurde in den wöchentlichen Sitzungen klar, dass eine Portierung der gesamten Bibliothek nicht möglich sein wird. Deshalb müssen gewisse Vereinfachungen vorgenommen werden, was allerdings ein Verständnis von UniCrypt Voraussetzt. Hätte ich mir von Anfang an die Zeit genommen, die Funktionsweise der Java Bibliothek genauer zu untersuchen, hätte ich früher festgestellt, dass das Framework gar nicht unabdingbar gewesen wäre. Ausserdem gibt es bereits verschiedene Ansätze ein klassenbasiertes Konzept in die JavaScript Welt bringen zu können. TypeScript [23] ist ein prominentes Beispiel dafür. Schlussendlich hatte das Framwork den Anspruch dessen Funktionalität zu imitieren, was ein zu ehrgeiziges Unterfangen gewesen wäre.

TypeScript benötigt allerdings einen Compiler, der reines JavaScript erzeugt. Im Vergleich dazu verwendet das Op Framwork von UniCryptJS ausschliesslich native JavaScript Funktionalität um die Typenprüfung vornehmen zu können.

2.2.2 Wurde die Zielvorgabe erfüllt?

Ich habe den Umfang und die Komplexität des Projektes unterschätzt. Anstelle früh direkt mit der Portierung zu beginnen, habe ich zu viel Zeit in das Framework investiert. Im Nachhinein wäre es sinnvoller gewesen TypeScript [23] oder etwas ähnliches zu verwenden. Das Framework deckt die grundlegenden Funktionalitäten ab und ist sehr kompakt <700 Zeilen Code. Die ElGamal Verschlüsselung und Entschlüsselung mithilfe der JavaScript Bibliothek funktioniert. Ein Pederson Commitment kann erstellt werden. Die Minimalanforderungen wurden eingehalten. Allerdings wäre das ursprüngliche Ziel gewesen, weitaus grössere Teile der Bibliothek zu portieren. Mithilfe des Frameworks ist dies nun zwar einfacher möglich. Ich habe also einen Lösungsvorschlag mit Proof-Of-Concept gelifert. TypeScript wäre aber aus meiner Sicht der bessere Weg.

3 Umgebung

3.1 Node.js

Zum entwickeln wurde Node.js als JavaScript runtime verwendet[18]. Dieses Programm verwendet "Google chrome V8 engine". Natürlich hätte UniCryptJS auch direkt mit Hilfe eines Browsers entwickelt werden können. Node.js verfügt über eine Vielzahl von Open Source Erweiterungen die mit npm (node package manager) verwaltet werden können.

Dies und die Bedienerfreundlichkeit machen Node.js für mich als Entwicklungsumgebung den Browsern weit überlegen.

3.2 Grunt.js

Grunt ist ein JavaScript task runner [15]. Die Software selbst macht nichts, sondern stellt eine Plattform für Pugins zur Verfügung. Mit deren Hilfe können Aufgaben spezifiziert werden, die ausgeführt werden sollen. Mühsame Arbeitsschritte die bisher von Hand ausgeführt werden mussten, lassen sich so automatisieren. Beispielsweise lassen sich verschiedene Textdateien zusammenfügen. Unittests können ausgeführt werden oder auch ein Webserver kann gestartet werden. Dies wird alles von Grunt automatisiert und kann mit einer Befehlszeile ausgeführt werden. Zur eigentlichen Entwicklung von UniCryptJS trägt Grunt nichts bei, allerdings vereinfacht es viele Aufgaben enorm.

3.3 Sublime Text 3

Der zur Entwicklung verwendete Texteditor [21]. Eine Syntaxhervorhebung für die meisten Sprachen wird unterstützt. Zusatzfunktionalitäten lassen sich per Package Control bequem installieren. Trotz sehr vieler nützlicher Funktionen und parametrierbaren Einstellungen handelt es sich bei Sublime Text 3 nicht um eine IDE.

3.4 Mocha

Mocha ist ein Test Framework[17]. Es läuft mit Node.js und im Browser. Ausserdem gibt es ein Grunt Plugin. Zusammen mit Expect.js[12], bildet diese Kombination die Testing-Suite für UniCryptJS. Expect.js ist eine minimalistisches BDD (Behavior Driven Development) assertion Software. Diese läuft ebenfalls in Node oder im Browser. Zusammen mit Mocha entstehen auch von aussenstehenden Personen leicht lesbare Tests.

3.5 Git

Der Code wird auf Github gehostet[27]. Git wird als Versionsverwaltungssoftware verwendet.

4 JavaScript

4.1 Funktionen, Objekte und Closures

JavaScirpt kennt verschiedene native Typen[22]. Allerdings sind es nicht besonders viele, verglichen mit Java. Die sechs Typen sind:

- undefined
- object
- boolean
- number
- string

- function

Besondere Aufmerksamkeit verdienen die Typen object und function. Es gibt in Java keine Entsprechung, trotzdem lassen sich einige Parallelen ziehen. Auf diesen ist das Framework aufgebaut, das ein Java ähnliches Verhalten simuliert.

4.1.1 Funktionen

JavaScirpt kennt Funktionen[13]. Diese gleichen syntaktisch den Methoden aus Java. Eine Funktion besteht aus einem Namen, null oder mehreren Parametern und einem Körper1.

```
function Funktion() {
}
```

Snippet 1: Minimale gültige JavaScript Funktion

Funktionen können definiert und anschliessend aufgerufen werden. Zusätzlich können sie ein Objekt erzeugen. Jede Funktion kann immer auf beide Arten verwendet werden Snippet 2. Jede Funktion hat eine Eigenschaft prototype. Diese ist für die Objekterzeugung von Bedeutung und kommt in Kombination mit dem "new" Operator zum Tragen.

```
var myFunction = function myFunction(x) {
        return x + 10;
}

var instance = new myFunction();
console.log(instance); // Log: myFunction {}
console.log(myFunction(10)); // Log: 20
```

Snippet 2: Funktionsaufruf und Objekterzeugung

In JavaScript sind Funktionen entweder Eigenschaften eines Objektes, Bestandteil eines functions scopes oder können einer Variable zugewiesen werden. Je nachdem wo und auf welche Art sie definiert werden, bestimmt dies welchem Objekt sie angehängt und wie sie sichtbar sind Snippet 3.

```
//Dem übergeordneten Host-Objekt zugewiesene Funktion
//In Node beispielsweise GLOBAL.myFunction()
//Im Browser window.myFunction()
function myFunction() {}

//Einer Variable zugewiesene Funktion
//Im Browser window.x.myFunction
var x = function myFunction() {}

//Eine Private Funktion die einer anderen Funktion zugewiesen ist. Existiert nur innerhalb d
//Lässt sich nicht von ausserhalb aufrufen.
function myFunction() {
        function myInnerFunction() {}
}

//Eine Funktion innerhalb eines Objektes. Ist über den Namen der Eigenschaft ansprechbar
//-> Assoziertes Array
// Resultiert im Browser in window.x.myFunction
x = {
        'myFunction': function myFunction(){}
}

//Vollständigkeit halber exotische Funktionsdefinitionsart
x = new Function('param1', 'param2', 'var variableInsideFunctionBody = param1 + param2; retu

console.log(x(10,20)) // Log: 30
```

Snippet 3: Verschiedene Funktionsdefinitionen

4.1.2 Closures

Das Scope gibt den Bereich an, in dem eine Variable oder eine Funktion sichtbar ist. Java kennt das Scope auf Klassenebene, in Methoden und in Schleifen. Eine Variable aus dem umgebenden Bereich ist innerhalb eines untergeordneten immer sichtbar. Das umgekehrte gilt nicht. Wird beispielsweise ein `boolean boolVal` auf Klassenebene deklariert, so kann auf diesen Wert innerhalb der Klassen-Methoden und wiederum innerhalb einer Schlaufe darauf zugegriffen werden.

JavaScript verhält sich anders. Eine Closure ist eine Funktion, die Referenzen zu unabhängigen Variablen im umgebenden Scope hat [9]. Sie erinnert sich also an die Umgebung, in der sie erstellt wurde. In dem Beispiel Snippet 4 wird das Verhalten sichtbar. In Java wäre die Variable "name" nur innerhalb der Methode sichtbar, in der sie deklariert worden ist. In der Zeile `var myFunc = makeFunc()` wird die Funktion makeFunc() ausgeführt und die Funktion displayName zurückgegeben. Zum Zeitpunkt der Aufforderung myFunc() könnte man erwarten, dass die Variable "name" bereits nicht mehr existiert. Dank dem Konzept von Closures(Funktionsabschlüsse) wird der String Mozilla von der Funktion displayName() gespeichert und später korrekt wiedergegeben.

```
function makeFunc() {
        var name = "Mozilla";
        function displayName() {
                return(name);
        }
        return displayName;
}

var myFunc = makeFunc();
console.log(myFunc()); // Log: Mozilla
```

Snippet 4: Closures Beispiel von MDN[9]

4.1.3 Objekte

Wird eine Funktion mit dem "new" Operator aufgerufen entsteht ein Objekt. Auch hier sind Closures von zentraler Bedeutung. Zusätzlich spielt auch die spezielle Eigenschaft "prototype" eine entscheidende Rolle. Durch den Aufruf mit "new" wird eine Funktion zum Konstruktor eines Objektes [11]. Auf dieses kann mit dem Schlüsselwort "this" zugegriffen werden. Üblicherweise und auch im Framework wird der Konstruktor dazu benutzt die Instanzvariablen zu deklarieren Snippet 5. JavaScript lässt allerdings jederzeit eine Variablendefinition zu. Anders als bei Java ist `instance.x = 10` selbst dann zulässig, wenn x zum Zeitpunkt der Objektgenerierung mit dem "new" Operator noch nicht deklariert wurde. Dies lässt sich mit einfrieren verhindern [19]. Funktionen, Variablen und Instanzvariablen die im Konstruktor definiert sind, werden für jedes daraus entstehende Objekt erneut im Memory abgelegt. Da die Funktionen aber für alle Objekte die mithilfe dieser Konstruktorfunktion erzeugt werden gleich bleiben, ist dies äusserst ineffizient. Der `function.prototype` hingegen liegt nur einmalig im Memory. Jedes davon abgeleitete Objekt teilt sich diesen Prototypen. Es ist also sinnvoll Funktionen generell direkt auf dem Prototypen zu definieren, anstatt im Konstruktor Snippet 5.

```
var functionToObject = function functionToObject() {
        this.x = 99;
        //Existiert für jedes Objekt im Memory
        //Funktioniert, ist aber nicht performant.
        this.instanceFunc = function instanceFunc() {
                return 'innerhalb des Konstruktors';
        }
}
//Exisitert nur auf dem prototypen. Diesen gibt es für alle Objekte nur einmal
//Obwohl beide Arten zugelassen sind, empfiehlt sich doch diese zu verwenden.
functionToObject.prototype.prototypeFunc = function prototypeFunc() {
        return 'Liegt auf dem prototype';
}
var newObject = new functionToObject();
console.log(newObject.constructor.name); // Log: functionToObject()
console.log(newObject.x); //Log: 99
console.log(newObject.instanceFunc()); // Log: innerhalb des Konstruktors
console.log(newObject.prototypeFunc()); // Log: Liegt auf dem prototype
```

Snippet 5: Konstruktoren und Funktionen resultieren in Objekten

4.2 Prototypen Vererbung

In JavaScript wird die Vererbung durch Verwendung des prototype erreicht [11]. Ein Objekt kann von einem anderen erben. Dazu wird der "prototype" der Tochter-Konstruktorfunktion auf ein Mutter-Objekt gesetzt. Dies führt dazu, dass die Tochter-Konstruktorfunktion überschrieben wird. Deshalb muss diese anschliessend wieder mit dem Prototypen verknüpft werden Snippet 6.

```
var myParentFunction = function myParentFunction() {
}

var myChildFunction = function myChildFunction() {
}

myChildFunction.prototype = new myParentFunction();
var instance1 = new myChildFunction();
console.log('Instance 1: ' + instance1.constructor.name);
myChildFunction.prototype.constructor = myChildFunction;
var instance2 = new myChildFunction();
console.log('Instance 2: ' + instance1.constructor.name);
```

Snippet 6: Prototypen basierte Vererbung

5 Liste der von UniCrypt benötigten Java Sprachkonstrukte

5.1 Compiler

5.1.1 Anforderung

Die IDE prüft zur Entwicklungszeit bereits, ob im Programmcode Fehler vorkommen. So lassen sich Probleme zur Laufzeit vorbeugen. Eclipse beispielsweise meldet bereits vor dem Compiliervorgang, wenn es eine fehlende Übereinstimmung der Typen gibt. Es wird bereits zum Entwicklungszeitpunkt weitgehend festgelegt, welche Variable welchen Typ aufweist. Generische Programmierung oder Interfaces lassen etwas Flexibilität zu.

Eine abstrakte Klasse lässt sich nicht instantiieren. Dies meldet bereits die IDE. Der Compiler scheitert und meldet einen Fehler.

JavaScript als Interpreter Sprache verfügt über keine solche Prüfungen zur Entwicklungszeit. Erst bei der Ausführung oder im Browser werden Fehler sichtbar. Da es sich um eine dynamisch typisierte Sprache handelt, die oft Konversionen automatisch durchführt, können falsche Berechnungen ausgeführt werden, ohne überhaupt eine Warnung zu erzeugen.

5.1.2 Implementierung in Javascript

Es ist nicht mögliche eine Prüfung zur Entwicklungszeit zu erreichen in JavaScript. Die Fehler werden immer erst zur Laufzeit sichtbar. Trotzdem kann verhindert werden, dass Probleme stillschweigend verborgen werden und der Code ein inkorrektes Resultat liefert. Verwendet der Programmierer Konstrukte, die in Java so nicht zugelassen wären, muss das Framework einen Fehler werfen. Dies geschieht mit `throw new Error('Errormessage');`

in JavaScript. Dies führt zu einem Abbruch der Ausführung durch einen Laufzeitfehler. Die Meldungen sind spezifisch und lassen sich leicht nachvollziehen. Stimmen beispielsweise die Typen einer Funktion nicht mit den Erwartungswerten überein, so wird künstlich eine Fehlermeldung erzeugt. Das Ziel ist es gewährleisten zu können, dass falls der Code ohne Probleme ausgeführt wurde, dass Resultat stimmt. Dies unter der Annahme, dass der Programm Abschnitt in UniCrypt funktioniert hat und gleichwertig übernommen wurde.

5.2 Klassen, Scope und Zugriffsmodifizierer

5.2.1 Anforderung an Klassen

Javascript kennt nur die Prototypen-Vererbung (ECMAScript 6 beinhaltet allerdings bereits ein natives Klassenkonzept[10]). Auch dieses Paradigma wird einer objektorientierten Programmierung zugeschrieben.

Eine Klasse in Java hat Attribute und Methoden. Die Attribute sind die Eigenschaften der Klasse. Sie beinhalten Objekte oder primitive Datentypen. Methoden bestimmen die Verhaltensweise der Klasse. Es werden Operationen definiert, die Attribute verwenden, zuordnen oder Variablen deklarieren.

Methoden in Java können überladen werden, je nach Anzahl und Typ der Parameter. Nun ist es allerdings möglich, nur eine der Überladungen in einer Unterklasse zu überschreiben. Ein Attribut darf nur bei der Definition der Klasse deklariert werden und nicht durch Methoden zu einem späteren Zeitpunkt.

5.2.2 Anforderungen Scope

In Java gibt es verschiedene Arten von Sichtbarkeit. Wird eine Variable im Klassenkontext deklariert, so ist sie immer der gesamten Klasse zugänglich (d.h. sie kann in Methoden und in Schleifen abgerufen werden). Wird eine Variable innerhalb einer Methode deklariert, so ist sie auch nur dort verfügbar. Ausserhalb der Methode kann nicht darauf zugegriffen werden. Dasselbe gilt für Variablen in einer Schlaufe. Diese sind in der Umgebenden Methode nicht sichtbar. Nach Abarbeiten der Schlaufe hören sie auf zu existieren und werden vom Garbage-Collector entsorgt.

5.2.3 Anforderungen Zugriffsmodifizierer

Attribute und Methoden einer Klasse können von außerhalb der Klasse eingesehen werden. Dies ist abhängig von der Sichtbarkeit des Attributes oder der Methode. Das Schlüsselwort "private" bestimmt, dass die Methode oder das Attribut ausschliesslich innerhalb der Klasse verfügbar sind. Das Schlüsselwort "protected" bedeutet, dass die Methode oder das Attribut innerhalb desselben Paketes verwendet werden kann. Ein Zugriffsmodifizierer "public" erlaubt den Zugriff von überall her, auch von ausserhalb der Klasse.

Das Schlüsselwort "static" erlaubt einer Methode von außerhalb der Klasse aufgerufen zu werden, ohne dass die Klasse instantiiert werden muss.

Mit "final" lässt sich eine Variable nur genau einmal initialisieren. Spätere Änderungen sind nicht mehr möglich. Eine Klasse mit final kann nicht ererbt werden. Eine "final" Methode darf nicht überschrieben werden.

5.2.4 Implementierung in Javascript

Das Konzept einer "static" Methode ist von entscheidender Bedeutung. Viele Klassen in UniCrypt haben einen privaten Konstruktor und eine statische getInstance() Methode. Deshalb muss das Framework dies abdecken.

Ich habe mich bewusst gegen eine Implementierung der Zugriffsmodifizierer entschieden. Per Konvention muss jede "private" Variable und Methode im Framework mit einem _ (underline Symbol) gekennzeichnet werden. Dies hat allerdings keine technischen Auswirkungen. Natürlich lässt sich nachträglich eine Entsprechung im Framework verankern, ohne den Code ändern zu müssen.

Es gibt verschiedene Ansätze eine private Variable oder eine private Methode in JavaScript zu definieren [4] [3]. Das entscheidende Argument gegen die technische Umsetzung in JavaScript war die Klarheit des Programmcodes. Das Ziel war einen Variablenzugriff wie in Java mit dem Punkt Operator erreichen zu können. Gewisse Techniken zur Privatisierung von Variablen verändern den Zugriff. Die Optionen zur rein internen Implementierung von privaten Members ohne Anpassung der Zugriffsart mit dem Punkt Operator würden das Framework weniger performant machen (Variablen und Methoden werden weiterhin nur mit dem _ Symbol gekennzeichnet, haben aber eine technische Relevanz).

Das Schlüsselwort "final" wird vom Framework nicht unterstützt.

5.3

5.4 Vererbung

5.4.1 Anforderung

Eine Klasse darf von einer und nur genau einer anderen erben. Dabei werden alle Variablen und alle Methoden der Mutterklasse in der Tochterklasse verfügbar. Mit dem Schlüsselwort "super" kann der Konstruktor der Mutterklasse aufgerufen werden. Es können damit aber auch Methoden der Mutterklasse aufgerufen werden, die von der Tochterklasse überschrieben wurden. Eine Klasse darf nur von einer abstrakten oder von einer normalen Klasse erben. Allerdings darf eine klasse mehrere Interfaces implementieren. Zu beachten sind bei der Vererbung verschiedene Konzepte. Methoden können überschrieben werden. Dazu wird eine identische Methode deklariert. Diese erhält nun den Vorzug gegenüber derjenigen, der Mutterklasse. Die zu überschreibende Methode darf nicht final sein. Gewisse Überprüfungen werden notwendig: Wurden die abstrakten Methoden implementiert? Darf die Klasse überhaupt vererbt werden (Mutterklasse nicht final)? Finale Klassen sind allerdings ein Konzept, dass bei Unicrypt nicht vorkommt und deshalb im voraus verworfen werden kann. Alle Attribute müssen zugänglich sein und alle Methoden vorhanden oder überschrieben werden.

5.4.2 Implementierung in Javascript

Damit nur von einer Mutterklasse geerbt werden kann, wird im Framework nach dem Schlüsselwort "extends" auch nur eine Klasse übergeben. Es kann eine Vererbungshierarchie entstehen. Damit Methoden der Tochterklasse diejenigen der Mutterklasse überschreiben, werden die Methoden zum endgültigen Objekt der Reihe nach von oben nach unten hinzugefügt. Da ein JavaScript Objekt ein assoziiertes Array darstellt, überschreibt eine Zuweisung zu einem bestehenden Schlüssel den vorhergehenden Wert. Da intern im Framework eine abstrakte und eine gewöhnliche Klasse denselben Programmcode teilen, kann auch eine abstrakte zur Mutterklasse erklärt werden. Bei der Definition einer Klasse

wird überprüft, ob alle abstrakten Methoden implementiert wurden. Falls dies nicht der Fall ist, wird die Klasse intern auf abstrakt gestellt, was eine Instantiierung verhindert.

Das Schlüsselwort "final" wird vom Framework auch auf Klassenebene nicht unterstützt.

5.5 Abstrakte Klassen

5.5.1 Anforderung

Abstrakte Klassen sind Klassen, die nicht instantiiert werden können. Sie sind erweiterte Interfaces oder umgekehrt ausgedrückt, sind Interfaces rein abstrakte Klassen [6]. Abstrakte Klassen können sowohl Methoden, wie auch reine Methodensignaturen definieren. Damit eine von einer abstrakten Klasse erbende Klasse nicht mehr abstrakt sein muss, müssen alle Methodensignaturen implementiert worden sein. Eine abstrakte Klasse verhält sich also wie ein Interface mit deklarierten Variablen und implementierten Methoden.

Es muss in JavaScript verhindert werden, dass eine abstrakte Klasse instantiiert wird. Sie muss vererbt werden können.

5.5.2 Implementierung in Javascript

Im Framework teilen sich die abstrakten und die gewöhnlichen Klassen denselben Code. Jede Klasse kennt den Zustand instantiierbar und abstrakt. Je nachdem ob alle abstrakten Methoden implementiert wurden, ist eine Transition möglich.

5.6 Interfaces

5.6.1 Anforderung

Die Anforderung an ein Interface ist eine Vereinfachung der Anforderung an abstrakte Klassen. Allerdings darf ein Interface ein anderes Interface durch 'extends' erweitern. Es muss also grundsätzlich zwischen Klasse, abstrakte Klasse und Interface unterschieden werden können bei der Vererbung.

5.6.2 Implementierung in Javascript

Interfaces werden vom Framework nur teilweise unterstützt. Zur Portierung von UniCrypt ist deren Verwendung nicht zwingend Voraussetzung, da es für jedes Interface eine abstrakte Klasse gibt, die dieses implementiert. Im Framework teilen Interfaces nicht den gleichen Code wie gewöhnliche und abstrakte Klassen. Ein Interface kann von mehreren erben. Diese werden in einem Array spezifiziert.

Die Klassen überprüfen bei der Instantiierung ob alle Methoden aus dem Interface implementiert wurde. Falls nicht, nimmt sie automatisch den Zustand Abstrakt an.

5.7 Typsicherheit und Generic

5.7.1 Anforderung

Java erlaubt durch Interfaces eine Referenz zu brauchen, die auf verschiedene Objekttypen zeigen kann. Allerdings wird dadurch sichergestellt, dass die aufzurufenden Methoden in

beiden Klassen vorkommen. In Javascript kann jedes Objekt in einer Variable enthalten sein. Erst der Methodenaufruf zeigt, ob die Funktion vorhanden ist oder nicht. Dies zerstört die Notwendigkeit von Interfaces. Von Hand lässt sich aber die Implementierung der Methoden prüfen oder auch das Vorhandensein des Interfaces. Dies soll Aufgabe des Frameworks sein. Auch sollen die nativen Typen angegeben werden können, wie Integer, String oder Zugehörigkeit zu einem gewissen Objekttypen.

Generische Klassen können unterschiedliche Typen aufnehmen. Eine generische Klasse in Java hat mehrere Optionen bei der Spezifizierung der Typen bei der Instantiierung. Bei der Invarianz kann ein spezifischer Datentyp angegeben werden. Bei der Kovarianz ist die Bedingung das jedes Element von einer spezifischen Klasse erbt. Bei der Kontravarianz muss jedes Element eine Superklasse einer spezifizierten Klasse sein. [14] Bei der Definition einer generischen Klasse kann angegeben werden, welches Interface die generischen Parameter implementieren müssen.

Eine Umsetzung sowohl der Typisierung wie auch der generischen Klassen bedeutet eine Restriktion (Spezialisierung) in Javascript, wohingegen diese Konzepte in Java eine Erweiterung (Generalisierung) darstellen.

5.7.2 Implementierung in Javascript

Um die Typenprüfung vornehmen zu können, wird im Framework ein Wrapper um die Methode gelegt. Dieser wird bei einem Funktionsaufruf zuerst aufgeführt. In diesem Wrapper werden die Parametertypen geprüft und erst bei Übereinstimmung wird der Inhalt der Methode ausgeführt.

Um zu beurteilen ob ein Objekt von einer bestimmten Klasse abstammt, wird der Name des Konstruktors verglichen. Falls diese nicht übereinstimmen wird in der "prototype chain" nach einer Entsprechung gefunden. Falls dieses Unterfangen erfolgreich ist, wird der Parameter akzeptiert, ansonsten wirft das Framework einen Fehler. Die generischen Typen werden bei der Instantiierung einmalig festgelegt und von dem Objekt intern in einem JavaScript-Objekt gehalten.

5.8 Unveränderlichkeit

5.8.1 Anforderung

Unicrypt setzt auf das Konzept der immutable objects. Sobald ein Objekt instantiiert wurde, gibt es keine Änderungen der Attribute mehr. Wird ein abgeändertes Objekt benötigt, so muss ein neues erstellt werden. Um dieses Konzept umsetzen zu können, müssen gewisse Design Richtlinien befolgt werden [20]:

- Keine setter Methoden. Es dürfen keine Objektreferenzen oder Werte verändert werden können
- möglichst alle Attribute als private und final deklarieren.
- Unterklassen sollten keine Methoden überschreiben dürfen. Entweder wird dabei die gesamte Klasse auf final gesetzt oder aber es wird mit Factory Methoden gearbeitet (Ansatz in Unicrypt). Wie bei einem Singleton wird der Konstruktor privat gemacht. Um eine neue Instanz zu erhalten muss eine "public" Methode `getInstance()` aufgerufen werden. Dabei müssen alle Parameter übergeben werden, da es keine setter Methoden gibt.
- Beinhaltet ein Attribut eine Referenz zu einem veränderbaren Objekt, so soll die Klasse keine Änderungsmethoden zur Verfügung stellen und nicht die Referenz des mutable object freigeben.

Damit ein solches Konzept umgesetzt werden kann, müssen gewisse Funktionen verfügbar sein. Ein Attribut muss als privat und als final deklariert werden können. Dasselbe gilt für den Konstruktor (mit der Implikation, dass das Objekt **nicht** instantiiert werden kann ausser über getInstance()).

5.8.2 Implementierung in Javascript

Hierbei handelt es sich um ein Design Pattern. Das Framework unterstützt die statische getInstance() Methode. Wird die Konvention mit dem _ underline Symbol eingehalten, so werden auch private Attribute definierbar. Ein Konstruktor kann als private deklariert werden. Mit diesen Voraussetzungen lässt sich dieses Design Pattern aus Java in JavaScript übernehmen.

6 Op Framework

6.1 Klassen

In Java lassen sich Klassen definieren. JavaScript unterstützt diese Funktionalität bisher noch nicht nativ. In ECMAScript 6 werden neue Keywords eingeführt, die eine Klassen basierte Programmierung unterstützen[10]. Da allerdings diese Spezifikation neu ist und noch von wenigen Browsern unterstützt wird, verwendet das Op Framework diese nicht. Stattdessen werden Klassendefinitionen in Prototypen basierte Vererbung umgewandelt.

```
Op.Class('Klassname', {
    //Deklarationsoptionen
}, {
    //Klassendefinition
});
```

Snippet 7: Klassendefinitionen

6.1.1 Klassenname

Der Klassenname bei der Deklaration ist ein einfacher String. Intern wird die Konstruktorfunktion umbenannt, so dass bei der Prüfung der Typen bei der Parameterübergabe die gesamte Prototypenkette untersucht werden kann.

6.1.2 Deklarationsoptionen

Das Objekt Deklarationsoptionen enthält Informationen zur Vererbung und zu generischen Typen. Die Eigenschaft 'extends' spezifiziert von welcher Klasse geerbt wird. 'implements' definiert ein Array aus Interfaces die implementiert werden müssen. 'generic' stellt ein Array aus Typen dar die zum Instanziierungszeitpunkt festgelegt werden müssen. Deklarationsoptionen sind optional. Sollten keine angegeben werden, kann als zweiter Parameter entweder null oder ein leeres Objekt übergeben werden.

Wie im Snippet 8 ersichtlich wird mit dem Schlüsselwort 'extends' der voll qualifizierende Name der Mutterklasse angegeben. Das Framework kümmert sich anschliessend um die

```
demo = {};
demo.BaseClass = Op.Class('BaseClass', {
},
{
    testFunction: function() {
        return 10;
    }
});
var ChildClass = Op.Class('ChildClass',  {
        'extends': demo.BaseClass
}, {
});
var instance = new ChildClass();
instance.testFunction(); // 10
```

Snippet 8: Klassendeklarationen

```
demo.Interface = Op.Interface('Interface', null, {
        funcOne: function() {

        }.paramType(['int']).returnType('string'),
});
demo.InterfaceClass = Op.Class('InterfaceClass', {
        'implements': [demo.Interface]
},{
        funcOne: function() {
                return 10;
        }.paramType(['int']).returnType('string')
});
```

Snippet 9: Implementierung von Interfaces

Vererbung.
Dasselbe gilt auch für die Implementierung von Interfaces Snippet 9.

Die Deklarationsoption 'generic' gibt die generischen Typen der Klasse an Snippet 10. Diese müssen bei der Instantiierung mit dem ersten Parameter typisiert werden. Die Eigenschaft 'generic' muss ein Array darstellen (Objekte sind nicht geordnet in JavaScript, Arrays hingegen schon). Innerhalb dieses Arrays dürfen per Konvention nur Großbuchstaben enthalten sein, die der Kondition `str.match(/^[A-Z]$/)` entsprechen. Eine Klasse kann also maximal 26 generische Parameter enthalten.

Die Eigenschaft 'extends' kennt eine zweite Form. Falls eine Kindklasse von einer generischen Mutterklasse erbt müssen die generischen Parameter im 'extends' Objekt spezifiziert werden. Dies kann entweder durch direkte Zuweisung eines Klassennamen oder durch Weitergabe der generischen Parameter erfolgen. Anders als in Java muss beachtet werden, dass für die selben Buchstaben immer auch das selbe Objekt verwendet werden muss. Dahingehend ist Snippet 11 nicht zulässig. In diesem Beispiel wäre trotz der Umkehrung der Reihenfolge der Parameter im 'extends' Objekt T und V in Funktionen aus beiden Klassen gleich. Wurde bei der Instantiierung von GenericChildClass T als string und V als int festgelegt, so gilt dieselbe Typenzuweisung auch für Funktionen, die in GenericClass deklariert wurden und eigentlich durch die Vertauschung eine Typisierung im 'extends' Objekt V als string und T als int erwarten würden.

Die Deklarationsoptionen entsprechen einer vereinfachten Verison der Klassendefinitionszeile in Java Snippet 12.

```
var GenericClass = Op.Class('GenericClass', {
    'generic': [
        'T', 'V'
    ]
},{
});
var GenericChildClass = Op.Class('GenericChildClass', {
    'generic': [
        'T', 'K'
    ],
    'extends': {
        'class' : GenericClass,
        'generic': [
            'T', 'string'
        ]
    }
},{
});
```

Snippet 10: Generische Typen

6.1.3 Klassendefinition

Die Klassendefinition ist ein Objekt mit allen Methoden und Variablen. Auf dem Objekt werden Eigenschaften definiert, die vom Op Framework in Methoden und Attribute umgewandelt werden. In Java entspricht dies dem eigentlichen Körper einer Klasse.

Die Eigenschaften sollten die in JavaScript übliche Syntax aufweisen Snippet 13. Das Framework bietet keine Handhabung für private Klassenmitglieder. Soll ein Attribut oder eine Methode den Zugriffsmodifizierer private erhalten, so wird der Variablenname mit einem vorgestellten _ (underline Symbol) versehen. Auch wenn das Framework dies nicht verhindert, so sollte doch per Konvention niemals auf einer Instanz etwas verwendet werden, dem ein _ vorgestellt ist. instance._privateAttribute = 4 ist also verboten, während (innerhalb einer Methode derselben Klasse) this._privateAttribute = 4 erlaubt ist.

Im Klassendefinitionsobjekt gibt es einige reservierte Schlüsselwörter. Diese magischen Keywords verhalten sich nicht wie erwartet, sondern weisen ein spezielles Verhalten auf. Niemals ist es möglich im Op.Framework eine Methode init, _init, $$super oder static zu definieren und diese als solche aufzurufen! instance.init() ist nicht zulässig (wirft einen Fehler).

Für den internen Gebrauch im Framework gibt es einige Framework-Variablen und Framework-Funktionen. Diese dürfen niemals verwendet werden. Auch dürfen in Klassendefinitionen keine solche definiert werden. Framwork spezifische Funktionen und Variablen folgen der Konvention _Variablenname_. Diese beginnen mit einem underline Symbol und enden auch mit einem solchen. Sollte eine solche Variable ausserhalb des Frameworks verändert werden, so wird das gesamte Programm vermutlich nicht mehr funktionieren und nur noch Fehler werfen.

```
var GenericClass = Op.Class('GenericClass', {
    'generic': [
        'T', 'V'
    ]
},{
});
var GenericChildClass = Op.Class('GenericChildClass', {
    'generic': [
        'T', 'V'
    ],
    'extends': {
        'class' : GenericClass,
        'generic': [
            'V', 'T' // Dürfen nicht vertauscht werden.
        ]
    }
},{
});
```

Snippet 11: Achtung: Nicht zulässige Deklaration; Ergibt nicht gewünschtes Resultat

```
public class GenericChildClass<E, V>
           extends GenericClass<E, V>
           implements InterfaceX<V> {
}
```

Snippet 12: Beispiel einer Klassendeklaration aus Java

```
// Funktioniert, da es der JavaScript Syntax entspricht
// Hat aber rein gar nichts mehr mit Java Syntax zu tun und ist deshalb nicht zu empfehlen
var obj = {};
obj['_x'] = 10;
obj['func'] = function(int1) {
    return int1 + this._x + this._privateFunc();
}.paramType(['int']).returnType('int');
obj._privateFunc = function() {
        return 10;
}.returnType('int');

var Class1 = Op.Class('BaseClass', null,obj);

//So wird eine Klasse richtig definiert
//Die Eigenschaften werden direkt in dem Objekt definiert
var Class2 = Op.Class('BaseClass', null,{
        _x: 10, //Per Konvention eine Private Variable
    func: function(int1) {
        return int1 + this._x + this._privateFunc();
    }.paramType(['int']).returnType('int'),
    _privateFunc: function() { // Per Konvention eine private Funktion
            return 10;
    }.returnType('int')
});

var c1 = new Class1();
var c2 = new Class2();
console.log('C1: ' + c1.func(10));
console.log('C2: ' + c2.func(10));
```

Snippet 13: Schlechte und gute Art eine Klassendefinition vorzunehmen

6.1.4 Konstruktor

Ein weiteres reserviertes Wort ist init. Kein Attribut sollte init genannt werden da es sich dabei zwingend um eine Methode handeln muss (JavaScript typeof init === 'function'). init repräsentiert den Konstruktor aus Java in JavaScript. Innerhalb der init Methode darf der Superkonstruktor aufgerufen werden mit this.$$super(). $$super ist ebenfalls Schlüsselwort. Er darf keine Methode mit diesem Namen definiert werden. Ist keine init Methode definiert wird der leere Standard Konstruktor verwendet. Dies ist im Framework wie folgt hinterlegt: obj.init = function init() Eine spezielle Form des Konstruktors ist die _init Methode. Dieses underline hat hier eine technische Bedeutung. Ein privater Konstruktor verhindert die Instantiierung der Klasse. Dies bedeutet, dass die Klasse eine statische getInstance() Methode enthalten muss, die eine Instanz zurück liefert. Dies ist eine weitere magische Methode. Wird sie als statisch definiert und nur dann, weist sie ein spezielles Verhalten auf. Wird das Schlüsselwort getInstance() verwendet, so kann die Klasse innerhalb und nur innerhalb dieser Methode instantiiert werden, sollte der Konstruktor privat sein 14. Im Framework wird nach Aufruf dieser die Blockade aufgehoben für die Ausführungsdauer.

Es dürfen nie gleichzeitig eine _init und eine init Methode definiert sein.

```
demo.PrivateConstructor = Op.Class('PrivateConstructor', null, {
        _init: function(constructorParam) {
                this.x = constructorParam;
        }.paramType(['int']),
        static: {
                getInstance: function() {
                        return new demo.PrivateConstructor(20);
                }
        },
        x: 10
});

var privateTester = demo.PrivateConstructor.getInstance();
//var fehlerWerfen = new demo.PrivateConstructor(20); // wirft einen Fehler wenn diese Zeile
console.log(privateTester.x); // Log: 20
```

Snippet 14: Ein privater Konstruktor wird definiert

6.1.5 Statische Attribute und Methoden

Enthält die Klassendefinition eine Eigenschaft static in Form eines Objektes, werden die so deklarierten Methoden und Attribute statisch verfügbar gemacht. Für den Aufruf einer statischen Methode muss der vollständig qualifizierende Name angegeben werden 15. Aus Gründen der Bequemlichkeit kann innerhalb der Methoden der Klasse mit this.static.methodenname direkt zugegriffen werden.

```
demo.StaticClass = Op.Class('StaticClass', null, {
        static: {
                z: 0,
                y: 0,
                increment: function() {
                        demo.StaticClass.z += 1;
                }
        },
        setY: function(y) {
                this.static.y = y;
        }
});

var staticClass = new demo.StaticClass();
demo.StaticClass.increment();
console.log('Z: ' + demo.StaticClass.z); // Z: 1
demo.StaticClass.increment();
console.log('Z: ' + demo.StaticClass.z); // Z: 2
staticClass.setY(10);
console.log('Y: ' + demo.StaticClass.y); // Y: 10
```

Snippet 15: Statische Attribute und Methoden

6.1.6 Funktionsdefinition und typen Sicherheit

JavaScript ist eine dynamisch typisierte Programmiersprache. Funktionen können erst zur Laufzeit den Typ ihrer Parameter und ihres Rückgabewertes kennen. Um dieses Konzept vom Framework abbilden zu können wurde der Funktionsprototyp erweitert.

Funktionen werden in der Klassendefinition als Eigenschaft definiert. Jede Funktion kann, muss aber nicht, die Typen der Parameter und des Rückgabewertes vorgeben. Hierbei kommen die Erweiterungen des Funktionsprototypen reutrnType und paramType zum Tragen Snippet 16. Falls die Parameter oder der Rückgabe Wert auf diese Weise spezifiziert wurden, übernimmt das Framework bei jedem Funktionsaufruf die Überprüfung der Typen. Auch dies kann erst zur Laufzeit stattfinden, wie bei Interpreter Sprachen üblich. Gegenüber der klassischen Verwendung von nicht typisierten Parametern bietet diese Vorgehensweise zwei Vorteile:

- Die Fehlermeldungen sind übersichtlicher
- In vielen Fällen konvertiert JavaScript nativ die Typen dynamisch ohne einen Fehler zu werfen. Daraus resultieren schwer einzuordnende, falsche Werte Snippet 17. Mit dem Framework kann dem Vorgebeugt werden.

```
func: function(param1, param2) {
        return param1 + param2;
}.paramType(['int', 'int']).returnType('int'),
```

Snippet 16: Beispiel einer typisierten Methode

Die Funktionserweiterung paramType nimmt als Parameter ein Array aus strings entgegen und returnType einen einzelnen string. Die Anzahl an Werten im Array für paramType entscheidet über die Anzahl an möglichen Argumenten für einen späteren Funktionsaufruf. Es können sowohl native Typen geprüft werden (wie Beispielsweise int, string(ist

```
func: function(param1, param2) {
        return param1 + param2
}
console.log('Ex1: ' + func(10, 20)); // Log: Ext1: 30
console.log('Ex2: ' + func(10)); // Log: Ext2: NaN
console.log('Ex3: ' + func(10, null)); // Log: Ext3: 10
console.log('Ex4: ' + func(10, '20')); // Log: Ext4: 1020
console.log('Ex5: ' + func(10, {x:20}.x)); // Log: Ext5: 30
```

Snippet 17: Unkontrollierbares Verhalten ohne Typisierung

in JavaScript keine Objekt, weshalb es klein geschrieben wird), long, array, boolean), als auch selbst definierte Klassen. Überprüft werden sowohl Anzahl an Parametern wie auch deren Typ. Bei der Angabe des Typs wird auch die Vererbungshierarchie berücksichtigt. Bei einem paramType(['ParentClass']) wird als Parameter eine Instanz der Klasse ChildClass akzeptiert, solange ChildClass von ParentClass erbt.

6.1.7 Funktionen überladen

Funktionen können überladen werden. JavaScript unterstützt diese Funktionalität nativ nicht. Durch die Typisierung der Parameter durch das Framework kann eine Überladung unterstützt werden. Ein JavaScript Objekt verhält sich wie ein associative array. Es findet eine Zuweisung 'Key' -> 'Value' statt. Es ist also nicht möglich zwei gleiche Schlüssel zu definieren. Da im Op Framework die Methoden als Eingeschaft eines Objektes abgelegt werden, können nicht zwei den gleichen Namen besitzen.

Um trotzdem überladen zu können, müssen die Bezeichnungen mit einer Nummer versehen werden. Diese müssen nicht zwingend in einer aufsteigenden Reihenfolge gewählt werden, entscheidend ist nur, dass die Methodennamen dadurch eindeutig werden (keine Zahl doppelt gewählt ist). Der Aufruf verhält sich schliesslich wie erwartet: instance.methodname(param1). Das Framework erkennt anhand der Parameter die gewünschte Methode und führt diese aus 18.

Eine Eigenheit des Frameworks ist, dass die Methoden immer noch über ihren ursprünglichen Namen ansprechbar sind. Diese beiden Schreibweisen führen zum selben Resultat: instance.method(singleIntParam) und instance.method1(singleIntParam). Solange die Definition von class.method1 = function(int1) .paramType(['int']) lautet.

```
var BaseClass = Op.Class('BaseClass', null,{
    func1: function(int1) {
        return int1;
    }.paramType(['int']).returnType('int'),
    func2: function(int1, int2) {
        return int1 + int2;
    }.paramType(['int','int']).returnType('int'),
    func2: function(int1, int2, string1) {
        return string1 + (int1 + int2);
    }.paramType(['int','int', 'string']).returnType('string'),
    func4: function(string1) {
        return string1 + ' : one single argument';
    }.paramType(['string']).returnType('string')
});

var ChildClass = Op.Class('ChildClass',  {
        'extends': BaseClass
}, {

});

var childClass = new ChildClass();
var result = childClass.func(10, 20, 'Result: ');
var result2 = childClass.func('Result: ');
var result3 = childClass.func1(10);
console.log('1: ' + result); // Log: 1: Result: 30
console.log('2: ' + result2); // Log: 2: Result:  : one single argument
console.log('3: ' + result3); // Log: 3: 10
```

Snippet 18: Überladung von Methoden

6.2 Generische Klassen

Das generische Konzept wurde auf Klassenebene realisiert, nicht aber bei generischen Methoden. Letzteres wird grundsätzlich nicht unterstützt, da es auch in UniCrypt nicht vorkommt.

Es werden in den Deklarationsoptionen die generischen Typen genannt und zum Instantiierungszeitpunkt festgelegt. Diese müssen zwingend aus nur einem Großbuchstaben bestehen und in einem Array abgelegt sein. Handelt es sich bei einer Klasse um eine generische Klasse, muss bei der Instantiierung als erster Parameter ein Array aus Typen dem Konstruktor übergeben werden, so dass die generische Klasse typisiert werden kann. Der erste Parameter für die init() Methode ist also beim Aufruf mit dem new Operator erst der zweite Parameter.

Die generischen Typen können fortan als Parametertyp angegeben werden und das Framework überprüft deren Korrektheit. Die Typen einer generischen Klasse können also auf zwei Arten festgelegt werden:

- Beim Aufruf durch den new Operator.
- Durch eine Tochterklasse in den Deklarationsoptionen mit der Eigenschaft 'extends': {'generic': ['typ']}

```
var BaseClass = Op.Class('BaseClass', {
        'generic': ['E', 'V'],
}, {
        x: null,
        init: function(val) {
                this.x = val;
        }.paramType(['E']),
        testFuncOne: function(val) {
                return this.x + val;
        }.paramType(['V']).returnType('E'),
        testFuncTwo: function(val) {
                return val + this.x;
        }.paramType(['V']).returnType('V'),
});

var ChildClass1 = Op.Class('ChildClass1', {
        'extends': {
                'class': BaseClass,
                'generic': ['string', 'int']
        }
}, {
        init: function(val) {
                this.$$super(val);
        }
});

var ChildClass2 = Op.Class('ChildClass2', {
        'generic': ['E', 'V'],
        'extends': {
                'class': BaseClass,
                'generic': ['E', 'V']
        }
}, {
        init: function(val) {
                this.$$super(val);
        }
});

var childClass1 = new ChildClass1('Nr. ');
var childClass2 = new ChildClass2(['int', 'string'],2);
console.log(childClass1.testFuncOne(1)); // Log: Nr. 1
console.log(childClass2.testFuncTwo('Nr. ')); // Log: Nr. 2
```

Snippet 19: Zwei Varianten zur Verwendung generischer Klassen

Das Konzept der generischen Typen ist für UniCrypt entscheidend. Auf einer hohen Abstraktionsebene, versucht die Bibliothek viele kryptographischen Operationen abbilden zu können. Klassen werden dazu generisch gehalten, sodass derselbe Programmcode für verschiedene Zwecke verwendet werden kann. Aus diesem Grund muss dieses Konzept zwingend vom Framework korrekt angewandt werden, um die Ähnlichkeit zur Java Bibliothek erreichen zu können.

6.3 Abstrakte Klassen

6.3.1 Deklaration

Java kennt abstrakte Klassen. Diese unterscheiden sich durch zwei wesentliche Punkte von gewöhnlichen[5]:

- Eine als abstrakt deklarierte Klasse kann nicht instantiiert werden
- Eine abstrakte Klasse kann keine oder mehrere abstrakte Methoden enthalten

Da eine gewöhnliche und eine abstrakte Klasse in allen übrigen Verhaltensmustern identisch sind, behandelt auch das Op Framwork die beiden mehrheitlich gleich. Anstatt des Aufrufs Op.Class('',{},{}) wird Op.AbstractClass('',{},{}) verwendet. Da es sich dabei nur um einen Wrapper handelt ergeben sich als Nebeneffekt zwei andere Arten eine Klasse als abstrakt zu definieren. Von diesen wird allerdings stark abgeraten.

In einer gewöhnlichen Klasse die mit Op.Class() definiert wurde, lassen sich ebenfalls abstrakte Methoden deklarieren. Ein vorgestelltes $ Symbol hat den zu erwartenden Effekt. Enthält eine Klasse eine abstrakte Methode wird sie automatisch abstrakt.

Aufgrund der Aufbauart vom Framework akzeptiert Op.Class('Klassenname',,,) einen vierten Parameter Snippet 20.

```
Op.Class('Klassname', {
    //Deklarationsoptionen
}, {
    //Klassendefinition
}, {
    //Optionen
    'abstract' = true;
});
```

Snippet 20: Definition von abstrakten Klassen wie im Framework

Wird bei einer normalen Klassendefinition ein Optionsobjekt mit 'abstract' : true angehängt so wird die Klasse ebenfalls abstrakt. Abstrakte Klassen können nicht instanziiert werden.

6.3.2 Abstrakte Methoden

Eine abstrakte Methode wird mit einem vorgestellten $ definiert. Ähnlich dem underline Symbol _ für private Variablen ist das $ Symbol eine Namenskonvention, die eingehalten werden muss. Das vorgestellte $ hingegen hat eine technische Wirkung. Methoden die auf diese Art benannt wurden, können nicht ausgeführt werden und müssen zwangsläufig von einer Tochterklasse überschrieben werden. Wird eine solche Methode nicht durch die Vererbungshierarchie an einer Stelle implementiert, so wirft das Framework einen Fehler bei der Instantiierung. Es wird allerdings nicht nach Parametern unterschieden, weshalb eine Überladung von abstrakten Methoden nicht möglich ist.

Während abstrakte Klassen ein entscheidendes Konzept von UniCrypt darstellen, so ist die Verwendung von abstrakten Methoden in UniCryptJS nicht notwendig. Der Vollständigkeit halber und zum sicherstellen, dass der Programmcode komplett übernommen wurde können sie allerdings hilfreich sein. Da JavaScript nicht zum Entwicklungszeitpunkt überprüft, ob alle aufgerufenen Methoden innerhalb einer Klasse auch tatsächlich

vorhanden sind, müssen in dem konkreten Fall von UniCrypt keine abstrakten Methoden deklariert worden sein.

Abstrakte Methoden werden nach dem Schema $Funktionsname -> Funktionsname überschrieben. Eine implementierende Methode hat kein vorgestelltes $ Symbol. Achtung: Attribute sind von dieser Namenskonvention nicht betroffen. Werden diese mit vorgestelltem $ Symbol definiert, so können diese normal verwendet werden. Op.Class('Test', null, $x: 5,) resultiert in einem gültigen instance.$x (ist gleich 5).

6.4 Interfaces

Interfaces weisen die gleiche Syntax wie Klassen auf. Variablendeklarationen werden aber ignoriert. Op.Interface(String Klassenname, Objekt Deklarationsoptionen, Objekt Interfacedefinition); Als Deklarationsoption wird das 'extends' Attribut verwendet. Da ein Interfaces allerdings mehrere Interfaces implementieren kann, wird hier anders als bei Klassen ein Array von Interfaces erwartet.

7 Framework Mängel und Probleme

7.1 TODO: Fehlende Funktionalität

7.1.1 Überladen von statischen Methoden

Statische Methoden können zum momentanen Zeitpunkt nicht überladen werden. Dies ist grundsätzlich kein grosses Problem, da dies in UniCrypt selten vorkommt. Eine statische Methode wird allerdings sehr oft überladen: getInstance(). Aus Bequemlichkeitsgründen ist es möglich eine Klasse mit diversen verschiedenen Eingabeparametern zu instantiieren. Um dies auch in UniCryptJS abbilden zu können, müsste auch das Op Framework eine Überladung sowohl der speziellen getInstance() wie auch der anderen statischen Methoden erlauben.

7.1.2 Überladen der Konstruktorfunktion

In der Regel wird in UniCrypt die getInstance() Methode überladen. Diese übernimmt die Parameterprüfung und ruft sich selbst so lange auf (die anderen Überladungen), bis alle Variablen vorhanden sind. Sobald alle notwendigen Werte mitgegeben oder erzeugt wurden, wird der private Konstruktor aufgerufen. Deshalb haben die meisten Klassen auch nur genau eine solche Methode.

Der Vollständigkeit halber wäre es trotzdem wichtig, dass diese Option besteht. Java unterstützt die Überladung von Konstruktoren und deshalb sollte auch das Op Framework diese Möglichkeit anbieten.

7.1.3 Einbeziehen von Interfaces bei der Typisierung

Das Op Framework unterstützt Interfaces. Da UniCrypt auf abstrakte Klassen setzt, ist deren Einsatz für eine erfolgreich Umsetzung von UniCryptJS nicht notwendig. Ich habe also bisher bewusst auf Interfaces verzichtet.

Das Op Framework besitzt allerdings diese Funktionalität. Trotzdem ist sie zum momentanen Zeitpunkt nur eingeschränkt nutzbar. Klassen können Interfaces implementieren.

Wird allerdings bei function(classThatImplementsCorrectInterface).paramType('interface'); ein Interface Namen angegeben, anstelle eines Klassennamens, so kann das Framework die Prüfung nicht durchführen. Der Parameter classThatImplementsCorrectInterface wird zurückgewiesen mit der Bemerkung, dass die Typen nicht übereinstimmen. Die Typisierung respektiert also die Vererbungshierarchie, nicht aber die Implementierungshierarchie.

7.2 Fehlende Funktionalität mit niedrigstufiger Priorität

7.2.1 Typisierung von Variablen

Variablen können innerhalb einer Methode nicht typisiert werden. Dieses Verhalten in Java abbilden zu können, ist nicht von entscheidender Bedeutung, solange die Parameter und der Rückgabewert typensicher sind. Auch deklariert als Attribute kann denn Variablen kein fixer Typ zugewiesen werden.

7.2.2 Zugriffsmodifizierer: Private, Protected und Public

Fast an jedem Ort im Java Code lässt sich ein private, protected oder public Schlüsselwort verwenden. Einige Beispiele:

- Klassenattribute
- Methoden (statisch und auf Instanz bezogen)
- Klassendefinitionen
- Innere Klassen

Von Interesse für das Framework sind die Attribute und die Methoden. Per Konvention werden diese mit einem underline Symbol gekennzeichnet. Dieses hat allerdings ausser bei der _init() Methode keine technische Relevanz. Selbstredend könnte diese Funktionalität nachträglich noch eingebaut werden. Es gibt allerdings wichtige Gründe die dagegen sprechen [4].

8 Testing

8.1 Testing mit Mocha

Zum Testen der Klassen und des Frameworks wurde Mocha [17] mit expect.js [12] verwendet. Die Syntax ist von aussenstehenden leicht verständlich. Es werden Testfälle beschrieben und vom Grunt Taskrunner [15] ausgeführt Snippet 21. In einem "before" hook werden die Initialbedingungen für den Testfall festgelegt. Darin sollten keine Laufzeitfehler auftauchen. Die einzelnen Unittests instantiieren die vorbereitete Klasse und testen das Resultat auf Erwartungswerte.

```
describe('Function overloading', function() {
    var BaseClass;
    before(function() {
        BaseClass = Op.Class('BaseClass', null,{
            func1: function(int1) {
                return int1;
            }.paramType(['int']).returnType('int'),
            func2: function(int1, int2) {
                return int1 + int2;
            }.paramType(['int','int']).returnType('int'),
            func3: function(int1, int2, string1) {
                return string1 + (int1 + int2);
            }.paramType(['int','int', 'string']).returnType('string'),
            func4: function(string1) {
                return string1 + ' one single argument';
            }.paramType(['string']).returnType('string')
        });
    });
    it('tests overload with int param', function() {
        var baseClass = new BaseClass();
        var result = baseClass.func(10);
        expect(result).to.be(10);
    });
    it('tests overload with string param', function() {
        var baseClass = new BaseClass();
        var result = baseClass.func('Just');
        expect(result).to.be('Just one single argument');
    });
    it('tests overload with double int param', function() {
        var baseClass = new BaseClass();
        var result = baseClass.func(10,40);
        expect(result).to.be(50);
    });
    it('tests overload with tripple param', function() {
        var baseClass = new BaseClass();
        var result = baseClass.func(10,40, 'Result: ');
        expect(result).to.be('Result: 50');
    });
});
```

Snippet 21: Auszug aus den Unittests

Table of Illustrations

Liste von Code Snippts

References

[1] Yusuf Aytaş. Achieving abstraction in javascript. http://www.yusufaytas.com/achieving-abstraction-in-javascript/, 05. 2013.

[2] Berner Fachhochschule BFH. Univote. https://www.univote.ch/voting-client/, 06. 2016.

[3] Tim Buschtöns. Four ways to deal with private members in javascript. http://eclipsesource.com/blogs/2013/07/05/private-members-in-javascript/, 07. 2013.

[4] Douglas Crockford. Private members in javascript. http://javascript.crockford.com/private.html, 06. 2001.

[5] Diverse. Abstract methods and classes. https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html, 06. 2016.

[6] Diverse. Abstrakte klasse. https://de.wikipedia.org/wiki/Abstrakte_Klasse, 06. 2016.

[7] Diverse. Abstrakte klasse. https://de.wikipedia.org/wiki/Compiler, 03. 2016.

[8] Diverse. Classes. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes, 06. 2016.

[9] Diverse. Closures). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures, 03. 2016.

[10] Diverse. Defining classes. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes, 03. 2016.

[11] Diverse. Details of the object model. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model, 06. 2016.

[12] Diverse. Expect. https://github.com/Automattic/expect.js, 06. 2016.

[13] Diverse. Functions. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions, 06. 2016.

[14] Diverse. Generische programmierung in java. https://de.wikipedia.org/wiki/Generische_Programmierung_in_Java, 03. 2016.

[15] Diverse. Grunt. http://gruntjs.com/, 06. 2016.

[16] Diverse. Javascript closures. http://www.w3schools.com/js/js_function_closures.asp, 03. 2016.

[17] Diverse. Mocha. https://mochajs.org/, 06. 2016.

[18] Diverse. Node.js. https://nodejs.org/en/, 06. 2016.

[19] Diverse. Object.freeze(). https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze, 06. 2016.

[20] Diverse. A strategy for defining immutable objects. https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html, 03. 2016.

[21] Diverse. Sublime text 3. https://www.sublimetext.com/3, 06. 2016.

[22] Diverse. typeof. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof, 06. 2016.

[23] Diverse. Typescript. https://www.typescriptlang.org/docs/handbook/generics.html, 06. 2016.

[24] Diverse. Unicrypt. https://github.com/bfh-evg/unicrypt, 06. 2016.

[25] Rob Gravelle. Extending javascript objects in the classical inheritance style. http://www.htmlgoodies.com/html5/javascript/extending-javascript-objects-in-the-classical-inheritance-style.htmlfbid=YPKqgryv5ex, 06. 2016.

[26] Azat Mardan. How to use mocha with node.js for test-driven development to avoid pain and ship products faster. http://webapplog.com/tdd/, 03. 2015.

[27] Marcel Portillo. Unicryptjs. https://github.com/editoral/UniCryptJS, 06. 2016.

[28] Norbert Schmidt. Javascript coding patterns: Objekt-vererbung mit prototypen. https://blog.mayflower.de/4451-JS-Vererbung-Prototypen.html, 12. 2013.

[29] toby ho. What is this thing called generators? http://tobyho.com/2013/06/16/what-are-generators/, 06. 2013.

[30] Philip Walton. Implementing private and protected members in javascript. http://philipwalton.com/articles/implementing-private-and-protected-members-in-javascript/, 04. 2014.

Selbständige Arbeit

Ich bestätige mit meiner Unterschrift, dass ich meine vorliegende Bachelor-Thesis selbständig durchgeführt habe. Alle Informationsquellen (Fachliteratur, Besprechungen mit Fachleuten, usw.) und anderen Hilfsmittel, die wesentlich zu meiner Arbeit beigetragen haben, sind in meinem Arbeitsbericht im Anhang vollständig aufgeführt. Sämtliche Inhalte, die nicht von mir stammen, sind mit dem genauen Hinweis auf ihre Quelle gekennzeichnet.

Ort, Datum:	Bern, 16.06.2016
Vorname, Nachname:	Marcel Portillo
Unterschrift:	.....