Managed Data in Javascript: An evaluation of its performance and practical applicability in web applications

Remco Matthijs van Buijtenen
s2714086

March 2018

Bachelor Thesis

University of Groningen
Supervisor: Tijs van der Storm

Contents

Abstract

We attempt to structure our programs according to a hierarchy that separates functionality into distinct parts (concerns) that make it easy to read, reason about and reuse our code. Certain forms of behaviour such as type-checking, logging and immutability cannot be separated properly using an object-oriented hierarchy. They occur in nearly every program and in many different parts of these programs. In an attempt to reduce the amount of code duplication for these cross-cutting concerns that are not key to our business logic, we implement Managed Data: a model driven approach that separates data structure from data semantics (behaviour). By keeping data structure separate from class semantics, we are able to introduce data behaviour in a more abstract way such that it is independent from the underlying data structure. This promotes code-reuse and improves the maintainability of our code. In an attempt to evaluate Managed Data in JavaScript, we'll measure performance and lines of code written compared to that of a regular implementation. We'll then evaluate the practical applicability of MD in the form of a single-page web application.

1 Introduction

In software design we attempt to structure our programs according to a form of hierarchy that makes it easier to read, understand, maintain and reason about our code. In object-oriented languages this comes down to a subdivision in packages/modules, classes and methods. We'd like to give every concept in our design its own place in order to limit the dependency on other code. However, there are many common features that are difficult if not impossible to keep separate. Think of type-checking, persistence, immutability, logging or synchronization through the use of API's. These so called *cross-cutting concerns*[1] defy the hierarchy that we use to structure our programs. They *cut across* multiple layers of abstractions. Therefore, they often require a slightly different implementation that is unique to its context. This is manageable during initial definition, but software is not a static thing. Instead it is subject to constant evolution; requirements change rapidly, user's expectations grow as new technologies become available and these same new technologies often offer a whole new set of solutions to an existing problem.

Let us take a look at a piece of code that implements one cross-cutting concern: immutability. A point, represented by x and y can be moved, but only if it is not locked. We can move locking behaviour to a separate class, but some of the locking semantics remain tied to the Point itself; each time before we mutate the points properties, we must check if the object is locked and provide an appropriate error (see figure 1). This type of implementation is not desirable, since each object that extends locking also inherits this part of the locking semantics.

```
1  class Locking {
2      lock() { this.locked = true }
3      unlock() { this.unlocked = true }
4  }
5
6
7  class Point extends Locking{
8      set_position(x, y){
9          if(this.locked) {
10             throw "The Point is locked!"
11         }
12         this.x = x
13         this.y = y
14     }
15 }
16
17 let p = new Point()
18 p.set_position(1,2)
19 p.lock()
20 p.set_position(2,2) // throws "The Point is locked!"
```

Figure 1: A class-based 2D point that implements immutability

For this small example it might seem like a trivial problem, but when a piece of software grows, so does the complexity of cross-cutting concerns due to their inherent entanglement with many different parts of the program. This leads to a real pain when it comes to maintainability, and when ignored for to long its maintenance can turn into a bottomless pit of time and resources. Cross-cutting concerns do not only affect the maintainability of existing code, but it also discourages code reuse, because it is often easier to copy and customize an existing solution to a similar problem, rather than to generalize the existing solution such that it is applicable to both problems.

Managed Data attempts to circumvent this by decoupling the structural definition of data from its class semantics in order to achieve a proper separation of concerns. This relies on three main components: 1) a language that describes the data's structure and relations: a **schema**, 2) a data manager that can create objects that match this description while adding semantics, and 3) integration with the host programming language by overriding the dot operator, so that managed data can be used in the same way as regular JavaScript objects [3]. So far Managed Data has been implemented in Java [3] and in Ruby. It has also been proposed for interpreted languages like JavaScript and Python [4]

Now, let us compare the example from figure 1 to a managed approach (figure 2). The point model is defined as a simple object with an x and a y coordinate, both of which must be a number. This description of structure, a *schema*, is written using the JSON schema syntax [6]. Locking semantics are separated into the Locking class like before, with the addition of a *set* method that will be called whenever we use the dot (.) operator to change a property. The DataManager takes care of attaching the Locking semantics to the *pointSchema*.

This form of model-driven development allows us to create programs that are highly cohesive and loosely coupled.[3] Behaviour of an existing data structure can be changed by attaching different data managers to it, and data managers are easily reused by adding them to new definitions.

In this paper we will first explain how managed data can be implemented in JavaScript using ECMA script 6's proxy API [5], together with the already widely known JSON schema[6] language for data definition. Next, we'll evaluate the performance of MD with that of a standard class-based ES6 implementation. Finally, we'll evaluate managed data within the context of *Single Page Applications* (SPAs) like the widely used AngularJS framework[11]. We'll do this by implementing a small web framework using the Model View Controller (MVC) architecture where Managed Data is the Model layer. The source of MD4JS, the example application used to measure performance, as well as the web framework can be found on github [2].

```
import DataManager

let pointSchema = {
    "name": "Point",
    "properties": {
        "x": {"type": "number"},
        "y": {"type": "number"}
}   }

let Locking = (superclass) => class extends superclass {
    lock() { this.locked = true }
    unlock() { this.unlocked = true }

    set(propKey, value) {
        if(this.locked) { throw "The ${this.klass} is locked!"}
        return super.set(propKey, value)
}   }

let manager = new DataManager(pointSchema, Locking)
let point = new DataManger.Point()
point.x = 2
point.lock()
point.y = 1 // throws "The Point is locked!"
```

Figure 2: A managed approach to the immutability cross-cutting concern

2 Managed Data

So what is managed data and how does it benefit us? Let us take another look at the 2D point from the previous section. We already mentioned that the goal is to separate structure from semantics as we did for locking. The DataManager is then able to generate an object for the defined schema, and attach the locking behaviour to it. Now, let's say we also want the point to log when it is mutated, as well as write any mutation directly to a database. For the regular implementation in figure 1 we'd need to revisit the Point class, and extend it such that it also implements this behaviour. With managed data however, there is no need to revisit the point's definition. It is just a pair of coordinates that shouldn't change. What we do want to change is the meaning of a point within the scope of our application. For this we can implement logging and persistence just like we did for immutability. For basic logging we can define a *set* method that tries to set a property and if it fails, we print an error message. When setting was successful, we also print a message that indicates this success. This implementation can be seen in figure 3.

```
1
2  let Logging = (superclass) => class extends superclass {
3      set(propKey, value) {
4          try { super.set(propKey, value) } catch (e) {
5              console.log('Exception when setting ${propKey} of ${
                   this.klass}'
6              throw e
7          }
8          console.log('set ${propKey}$ of $(this.klass) to ${value}')
9      }
10 }
```

Figure 3: Implementation of the logging cross-cutting concern.

Persistence can be implemented using the same principle. When setting a property, we first attempt to change the properties' value, and then we write the change to the database. For now we consider this database to be a black box which returns a unique identifier upon record creation, and requires a JSON object that contains this id to write to the database. This implementation can be seen in figure 4.

```
1  import Database
2  let Persistence = (superclass) => class extends superclass {
3      init(inits) {
4          super.init(inits)
5          this.id = Database.create(this)
6      }
7
8      set(propKey, value) {
9          super.set(propKey, value)
10         let d = {"id": this.id}
11         d[propKey] = value
12         Database.update(d)
13     }
14 }
```

Figure 4: Managed implementation of persistence using a simple database.

With two more sets of behaviour, we can now use the DataManager to mix and match the semantics of our point. Figure 5 illustrates how various cross-cutting concerns can be added to the point. Logging can be added, and it will log changes made to the point. We can create another one with persistence, and all it will do is write its state to the database. A third point is created with Logging, Persistence and Locking behaviour. All three instances use the exact same definition of a point, but their semantics vary greatly because of the way the DataManager dynamically adds the behaviour to the point.

```
import DataManager
import pointSchema
import Locking

loggingManager = new DataManager(pointSchema, Logging)
syncManager = new DataManager(pointSchema, Persistence)
loggingSyncManager = new DataManager(pointSchema, Logging,
    Persistence, Locking)


p1 = new loggingManager.Point()
p1.x = 42 // prints "set 'x' of 'Point' to 42

p2 = new syncManager.Point()
p2.x = 42 // writes x = 42 to database

p3 = new syncManager.Point()
p3.x = 42 // logs value of x and writes x = 42 to database
p3.lock()
p3.y = 2
// throws exception that the point is locked
// logs: "The Point is locked!"
// does not write x to the database
```

Figure 5: Using a DataManager to create three points with varying semantics while all are structurally equal

3 Schemas

The point was defined as a JSON (JavaScript Object Notation) schema[6]. This schema is a widely used standard for the definition of data structures, formatting constraints and exact values. Most commonly it is used for data validation during serialization/de-serialization, as well as for user input (form) validation. Many tools exist that implement JSON schema validation and even though performance varies widely[8], they're all based on the offical JSON schema validation draft. However, none of these validators support the capability to validate individual *fields* of a data-structure. They only operate on entire objects.

We decided to make JSON schema's type validation mandatory in MD4JS because it adds an important invariant to the program: at any given moment during execution we can be certain that the state of our program is exactly matching the defined schema. This helps with the implementation of relations between managed objects. So when class A defines a one-to-one relation with class B, for any given instance of A with a relation to B we can be certain that the object on the other side is indeed an instance of B, and thus also implements B's management behaviour. So when A is modified in such a way that B must also be updated, then A can outsource the modification of B to B's data manager. This prevents the overlap of A's scope into that of B and thus allows the classes to implement independent behaviour for a single operation. Because it is not desirable to have an implementation that is only strictly typed in some cases, we

decided to implement type validation as a mandatory requirement for managed objects.

Because MD4JS is strictly typed, we must be able to ensure that each individual field matches the given schema at all times. Matching an entire data-structure against its schema is fast, but not fast enough to do continuously at run-time[8]. Moreover, even if this was a viable option, we'd waste a lot of computing power because for any given assignment to a field, all other fields must be validated as well. Therefore, a data-structure that supports run-time validation is needed. Instead, it is more efficient to use JavaScript's O(1) sting based indexing for objects.

In order to explain the concept of schemas for data representation, we'll use the example of a simple state machine. We can define a state machine as a set of *states*, one of which is the start state. We'll also give each machine a human-readable name. A state is a collection of incoming and outgoing *transitions*. A transition is described by an event and a pair of states that we'll call *from* and *to*. When the described event occurs we can use this transition to go to the next state. This definition of a state machine is illustrated in the class diagram in figure 6. As you can see, a state-machine has 0 or more states, while a state always belongs to at most 1 machine. A state always has 0 or more transitions and a transition belongs to exactly two states named *from* and *to*.

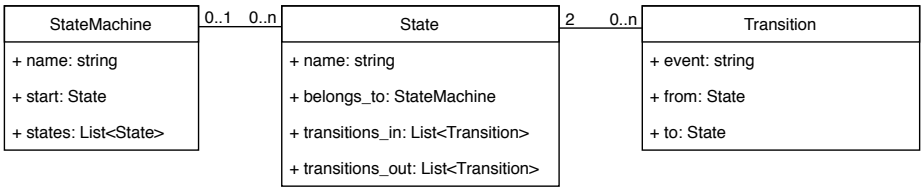

Figure 6: Class diagram for a state machine.

For this particular example, we'll use a state machine that represents a door (figure 7). This door can either be open or closed, and therefore it has two states. Furthermore, this door has two transitions; when the door is open, it can be closed and when the door is closed, it can be opened.

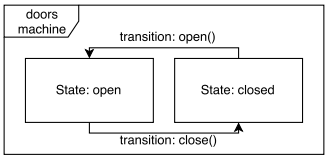

Figure 7: The doors state machine. States are represented as boxes and transitions are the arrows that connect them.

Now that we've defined what our state machine should look like, its class diagram can be translated into a schema using three steps: 1) define the classes,

2) add properties and 3) add related fields that match the arity defined in the class diagram. We'll begin by defining the classes. At the top, we have a single main object: the state machine itself. This will form the entry-point of the program. To which definitions can be added. Essentially, a definition is a separate schema that shares its meta information with the schema that it belongs to. We can thus define additional classes in the definitions section. In our case we will add both the State and the Transition here. Figure 8 contains a schema that defines the 3 classes required for a state machine.

```
let machineSchema = {
    "name": "StateMachine",
    "definitions": {
        "State": {},
        "Transition": {}
}   }
```

Figure 8: Defining the classes from figure 6 in JSON schema

The next step is to add all properties from the class diagram that are made up of primitive types like strings and numbers. For the StateMachine and State classes we have the *name* property and for the Transition we have an "event". All 3 have *string* as their type. We add these to the *properties* keyword in the schema, just like the point's x and y properties in figure 2.

Finally, object relations must be added to the schema. MD4JS supports four types of relations: one-way (where no inverse accessor is generated), one-to-one, one-to-many and many-to-many. We can add the *start* field as a simple one-way relation from StateMachine to State:

```
    {"start": {"type": "object", "$ref": "#/definitions/State"}}
```

The *$ref* keyword tells the data manager where to locate the definition of the referred class. Next, we'll add the *states* property. This property's arity matches that of a one-to-many relation. We'll also add inverse accessor so the State's machine can be accessed through the *belongs_to* key. For this we define a one-to-many relation from StateMachine to State and add it to StateMachine's relations property.

```
    "relations": {
        "oneToMany": [{
            "$ref": "#/definitions/State",
            "referrer": "states",
            "referred": "belongs_to"
        }]
    }
```

Figure 9: One to many relation from StateMachine to State

The relation defined in figure 9 will result into 2 fields being generated: a field *states* on StateMachine of type Array<State> and a field *belongs_to* on State of type StateMachine. We'll add two more one-to-many relations from State to Transition for the transitions_in and transitions_out fields and that concludes the schema. Figure 10 shows how the Klass definitions, properties and relations come together to form a full JSON schema defining a state machine.

```
let machineSchema = {
    "name": "StateMachine",
    "properties": {
        'name': {'type': 'string'},
        'start': {'type': 'object', '$ref': '#/definitions/
            State'}
    },
    "relations": {
        "oneToMany": [{
            "$ref": "#/definitions/State",
            "referrer": "states",
            "referred": "belongs_to"
        }]
    },
    "definitions": {
        "State": {
            "properties": {
                'name': {'type': 'string'}
            },
            "relations":{
                "oneToMany": [{
                    "$ref": "#/definitions/Transition",
                    "referrer": "transition_in",
                    "referred": "to"
                }, {
                    "$ref": "#/definitions/Transition",
                    "referrer": "transitions_out",
                    "referred": "from"
                }]
            }
        },
        "Transition": {
            "properties": {
                "event": {"type": "string"},
            "relations":{}
        }
    }
```

Figure 10: StateMachine schema after adding classes, properties and relations

A DataManager is able to transform this schema such that we can create a door. Figure 11 does this by initializing a new DataManager using the machineSchema. This DataManager can then create the two states that a door can have. Next, two transitions are created that link the two states together in a cycle. Note that because of the oneToMany relations that were defined earlier, it is not necessary to also add the transitions to the states. This can be done

automatically because an inverse field was defined using the *referred* key.

```
import DataManager
import machineSchema

manager = new DataManager(machineSchema)

door = new manager.StateMachine({"name": "doors"})
opened = new manager.State({"name": "opened"})
closed = new manager.State({"name": "closed"})
open = new manager.Transition({"from": closed, "to": opened, "
    event": "open"},
close = new manager.Transition({"from : opened, "to": closed, "
    event": "close"},

// closed.transitions_out == [open]
// open.from == closed && open.to == opened
```

Figure 11: Creating a door with two states and one transition using the machineSchema and a DataManager

3.1 Parsing the schema

With the basics of JSON schema out of the way, we can begin converting it into a structure that we can use to validate the fields of managed objects. We'll iterate through the schema twice: the first pass identifies Klass definitions and stores a reference to their location. The second pass will then identify each Klass's properties and store their type and value constraints. During this step relations are also identified, and fields are generated for both the related field and its inverse. For this we use the data about definitions that was gathered during the first pass. We store this information in two data structures: a class *KlassSchema*, which holds information about the Klass's fields and types. The field information is the JSON schema representation of that field. A class *Schema* maps klass names to KlassSchemas. This class is used to determine which schema should be used to construct a managed object. Both classes also contain methods that make up a basic interface for the MObject and MField, such that fast validation of structure and types can be performed (JavaScript's string based indexing of objects uses hashing for O(1) lookup). This architecture is illustrated in the class diagram of figure 12.

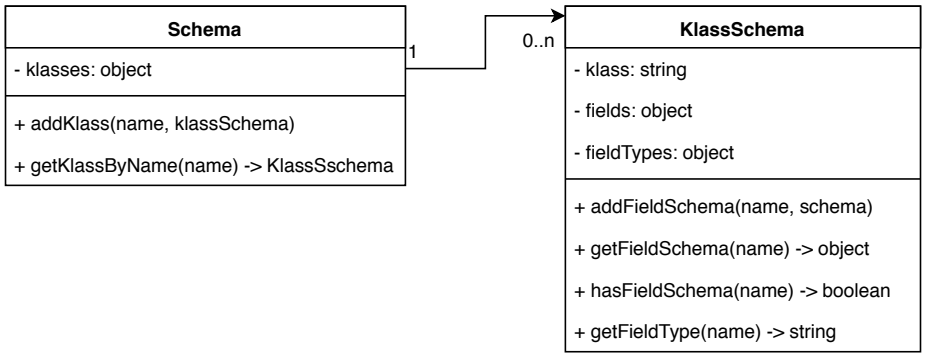

Figure 12: Class diagram that illustrates the structure of *Schema* and *KlassSchema*

```
//resulting field for klass StateMachine:
{
    "type": "array",
    "key": "states",
    "schema": {
    "type": "array",
    "items": [{"type": "object", "klass": "State"}],
    "inverseKey": "machine",
    "inverseType": "object"
  }
}

//resulting field for klass State:
{
    "type": "object",
    "key": "belongs_to",
    "schema": {
        "type": "object",
        "klass": "StateMachine",
        "inverseKey": "states",
        "inverseType": "array"
    }
}
```

Figure 13: Converting a one-to-many relation into two field schemas

Finally, we iterate through the relations defined under the 'relations' key, and use the paths that we extracted earlier to create the appropriate field schemas for these relations. If an object refers to many, we create an array field of type *object* where each item of the array must be an instance of the Klass defined in the related schema. If the object refers to a single object, we'll add an object field of the current object's class to the related schema under the inverse name that can be used to access this field. Essentially, a one-to-one relation results in two one-way related objects. A one-to-many results in a collection of one-way

references on one side, and a single one-way reference on the other. The many-to-many relation results in a collection of one-way references on both objects.

The generated field schemas for the oneToMany relation from figure 9 can be seen in figure 13. This field is then stored in the KlassSchema's *fields* property. Each field, including primitive types, consists of three properties:

- A type; the string representation of a JavaScript primitive.
- A key; a string that can be used to access the field
- A schema; the JSONM representation of the field, extended with relational and klass information

4 Managed Objects

The next challenge that we need to overcome is the design of a **Managed Object**. At the very least, a managed object must:

1. Ensure that data can be stored and retrieved according to the schema definition. Therefore, a managed object must do type checking at run-time. This also holds for the *one-way, one-to-one, one-to-many* and *many-to-many* relations.
2. Be integrated with the host programming language, such that it can be used as any other object.
3. Provide a way for programmers to implement custom management behaviour.

4.1 Storing data: Managed Fields

Whenever an attempt is made to read or write to a property on a managed object we must be able to ensure that management behaviour is applied to the property. By default, the management behaviour of a single property consists of two parts: structural validation and type/value validation. For the first, if a property is requested, we can query the schema about the property, and if it exists we continue. This behaviour is the same for each property regardless of its type, and we can thus implement this on the managed object. However, validating the basic type and value constraints of JSON schema requires an implementation that is unique to its type. For this we introduce *Managed Fields*. A managed field manages a single property and performs type checking for this property at run-time, ensuring it state matches the definition in the schema at all times. These primitive types can be further subdivided into two categories: simple fields and complex fields. A simple field consists of a single type only, while complex types can have values or collections that consist of one or multiple types. Below are the specific types and to which category they belong:

simple fields

- integer fields
- number fields
- string fields
- boolean fields
- related object fields

complex fields

- array fields (collections)
- enums
- oneOf fields (basically an enum for objects)

Each field provides a *setValue* and *getValue* method for retrieving and mutating data. The setValue method calls a *validate* method that matches the given value against the schema, and if it is valid it proceeds with changing the value. This makes up the basic MField class, from which all other fields will inherit. They can then extend the validate method to implement behaviour that is unique to its type.

```
1  class MField {
2      constructor(schema, type, initialValue) {
3          this.schema = schema
4          this.type = type
5          this.value = initialValue
6      }
7
8      getValue() {
9          return this.value
10     }
11
12     setValue(value) {
13         let valid, error = this.validate(value)
14         if(valid) {
15             this.value = value
16         } else {
17             throw error
18         }
19     }
20
21     validate(value) {
22         // perform validation for this specific type based on this.
               schema
23     }
24 }
```

Figure 14: The basic definition of a managed field

4.2 The MObject

Figure 14 illustrates the basic API that each MField provides to the MObject: a way to set values, a way to retrieve values and a constructor that receives a schema, a type and an optional initial value. With the definition of managed fields out of the way, we can move on to the managed object. Based on a given schema it is possible to construct a corresponding field for a property using the factory pattern. We define a factory method that when given a schema for a single property returns an instance of the corresponding field class. When the managed object is created we iterate through the schema and create the corresponding field for each property by invoking the MFieldFactory as is illustrated in figure 15. This factory takes a type, and returns the corresponding MField instance.

Similarly, we can define a *get* and *set* method that validates structure and then invokes the corresponding *setValue* and *getValue* on the managed field. Assuming that these methods receive a string that indicates which property is requested, we can validate structure by checking whether this property is present in the schema. If so, we pass it to the MField and otherwise we throw an error. This can also be seen in figure 15.

```
function MFieldFactory(schema, init) {
    switch(schema.type) {
        case "string": {
            return StringMField(schema, init)
        }
        // cases for all other primitives
    }
}
class MObject() {
    constructor(schema) {
        this.schema = schema
        this.data = {}
    }

    init(inits) {
        for(let property in this.schema) {
            this.data[property] = MFieldFactory(this.schema[
                property], inits[property])
        }
    }

    get(propKey) {
        if(propKey in this.schema) {
            return this.data[propKey].getValue()
        } else {
            throw "Property ${propKey} does not exist"
        }
    }

    def set(propKey, value) {
        if(propKey in this.schema) {
            return this.data[propKey].setValue(value)
        } else {
            throw "Property ${propKey} does not exist"
        }
    }
}
```

Figure 15: A basic managed object that initializes managed fields and provides a way to retrieve and set their value

These get and set methods should now be invoked whenever the dot operator is used on a managed object. Furthermore, we want to capture and pass the name of the requested property to these methods as well. For this ES6 implements proxies. A proxy is an object that transparently wraps around another object in order to capture events that happen to the wrapped object. Think of events like constructor calls, getting properties, setting properties, function calls and more. The captured event can then be modified before it is passed to the wrapped object. Let's return to the point example once again. We defined this point using the Cartesian coordinate system, but let's say we want to use the point as if it were a polar coordinate. We can define a proxy that transforms polar coordinates into Cartesian before retrieving its position by trapping the

get invocation of a proper. Figure 16 redefines the point as well as a proxy that performs the transformation between coordinate systems. A point can then be created by giving it an x and a y value. When the method getPosition is retrieved the proxy will intercept it, returning a function that converts the result of getPosition to polar:

```
class PointHandler = {
    get(target, key, receiver) {
        if(key == "position") {
            return function () {
                return cart2polar(target.getPosition())
            }
        } else {
            return target[key]
        }
    }
}

class Point {
    constructor(x, y) {
        this.position = [x, y]
    }

    getPosition() {
        return this.position
    }
}

p = new Point(3, 4)
console.log(p.position) // prints {"r": 5, "phi": 0.75}
```

Figure 16: Cartesian to polar coordinates using ES6 proxies

5 DataManagers

The last step in our design is the data manager. A data manager must implement at least the following: 1) accept a schema, and transform this into a data structure that can be used by the managed objects that it creates; 2) Be able to create new managed objects that match the schema definition and apply a set of mixins to it, and 3) It should be integrated with the host. This integration will form the second proxy layer.

5.1 Extending semantics through mixins

It should be possible to extend the behaviour of the basic managed objects. This is the place where cross-cutting concerns can be implemented in such a way that they work for any object. Initially, classical inheritance comes to mind. This approach works, but it discourages code reuse, since data managers would the become dependent on their predecessor. We'd like to be able to use a single

concern in combination with any other concern, in any order we'd like. Luckily, interpreted languages such as JavaScript, Ruby and Python provide a solution to this: **mixins**. A mixin is a class that is used to implement multiple inheritance for objects. Because JavaScript does not implement true inheritance, but rather uses a chain of prototypes. We can define a mixin as a function that accepts a superclass, and returns an anonymous class that extends the given superclass like so:

```
let mixin = (superclass) => class extends superclass {
    // override method
    method(arg) {
        // do something
    }
}
```

Figure 17: Definition of a mixin

This approach allows the dynamic extension of classes at runtime. This is exactly what we want the DataManager to do. It is thus possible to create a new class given a base class and a list of mixins by repeatedly applying the mixin function to the baseclass:

```
log = (superclass) => class extends superclass {
    m() {
        v = super.m();
        console.log(`v = ${v}`)
        return v
    }
}

save = superclass => class extends superclass {
    m() {
        v = super.m()
        localstorage['v'] = v
        return v}
}

class Base {
    def m() {return 1}
}
let mixins = [log, save]

finalClass = mixins.reduce((base, next) => {return next(base)},
    Base)
result = new finalClass.m()
// prints: "v = 1"
// localstorage[v] == 1
// result == 1
```

Figure 18: Applying two mixins to a base class

We can demonstrate this by looking at the inheritance chain for a single method as a two-way pipeline. The parameters come in through the method defined in the top-most mixin, and while traveling down to the managed field, each mixin can make modifications to the request. The managed field handles the request and then sends it back through the pipeline in reverse order. Here, post-processing of the returned data can be implemented. See figure 19 for an illustration of this pipeline.

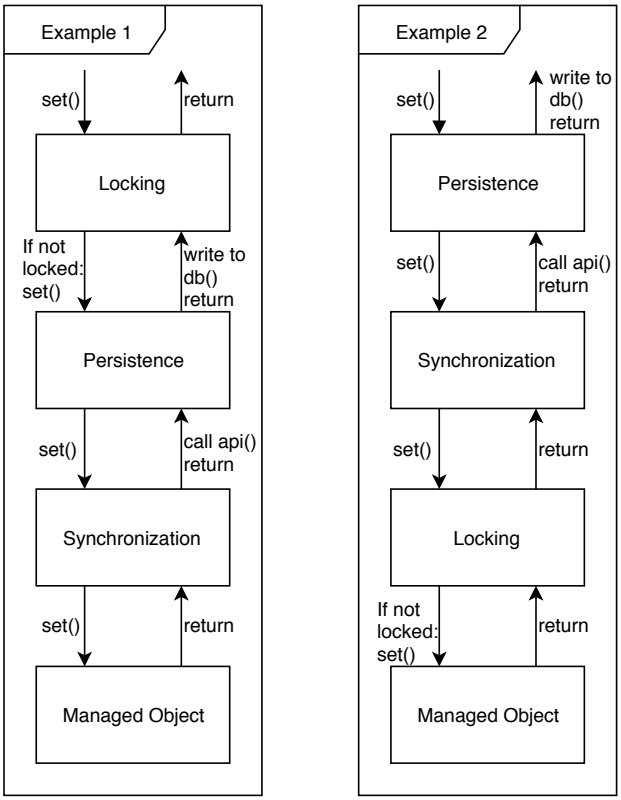

Figure 19: Illustration of a mixin pipeline where the inheritance does not matter.

5.2 MObject factory

We now have all the ingredients for a factory for managed objects: a schema, a managed object and way to add semantics to the managed object through mixins. Upon initialization of the DataManager we can generate a method for each Klass defined in the Schema. These factory methods receive the corresponding Klass's schema, as well as all mixins that should be added to it by binding it to the methods *this* pseudo-variable. A new MObject can then be created by applying all mixins to the base MObject. The next step is to wrap the MObject in a proxy similar to that of figure 16. We want to trap both *get* and *set* set

events that were invoked by usage of both the *dot* operator and string based indexing. The factory method itself should also be wrapped in a proxy; one that traps constructor calls. We then redirect this call to the corresponding factory method on the DataManager. See Appendix B for the full implementation of these proxies. This description of the factory method and initialization routine for the DataManager is shown in figure 20. A mockup of the method parseSchema has been added. This represents a recursive descent parser that traverses the given JSON schema. The full implementation can be found on github [2].

```
import Schema, KlassSchema
import FactoryHandler

parseSchema(schema) {
    let schema = new Schema()
    // 1. extract Klasses
    // 2. extract properties
    // 3. parse relations
    return schema
}

class DataManager{
    constructor(schema, ...mixins) {
    this.mixins = mixins;
    this.schema = parseSchema(schema);

    for(let klass in this.schema.klasses) {
        /* add a proxied factory with all relevant data bound to 'this' */
      this[klass] = new Proxy(this.factory.bind({
        schema: this.schema.getKlassByName(klass),
        mixins: this.mixins,
        klass: klass,
      }), new FactoryHandler())
    }
  }

  factory(inits) {
      mobjClass = (class extends mix(MObject).with(...this.mixins) {});

    let mobj = new mobjClass(this.schema);
    let mObjProxy = new Proxy(mobj, new MObjectHandler());

    /* The MObject needs a pointer to its own proxy for when an inverse field is found */
    mObjProxy.setThisProxy(mObjProxy);
    mObjProxy.init(inits);

    return mObjProxy;
  }
}
```

Figure 20: A minimal implementation of the DataManager

6 Putting it all together

With all the components in place, we can take a look at the overall architecture of MD4JS. Figure **??** contains a class diagram of what has been covered so far. The Schema and KlassSchema are made by the DataManager by calling the recursive descent parser. The DataManager then creates the MObject which initializes the MField based on the schema. Note that only the base MField has been included, extending classes have been omitted for brevity. Both the

DataManager's factory method and the MObject are wrapped by the appropriate handlers. The methods that make up the interface of MD4JS have been marked with a +. The two proxies make up the integration part of the interface: they override the dot and new operators, while the Schema, KlassSchema and MField create an interface where mixins can interact with data and structural information about the data.

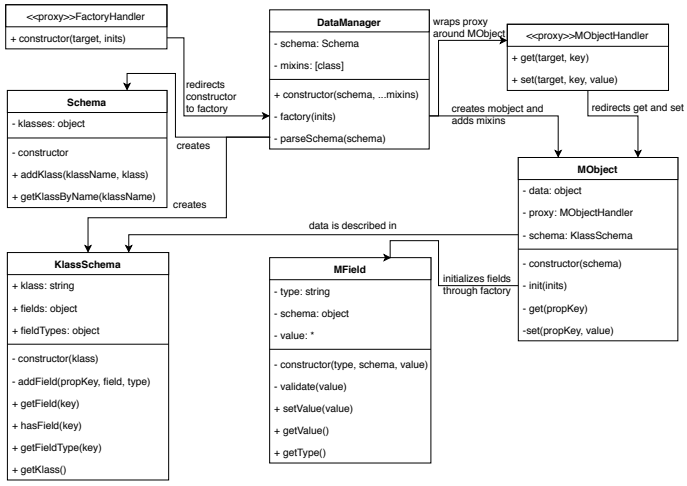

Figure 21: Class diagram of the architecture of MD4JS. Methods and properties marked with a + make u the interface for managed data.

6.1 Running the door state-machine

With all components defined, the doors state machine can be initalized by creating a new StateMachine with two states, *opened* and *closed*, as well as two transitions between them. Upon creation of the transitions, we pass them two states. The transitions will then add themselves to the corresponding *transitions_in* and *transitions_out* properties on the state because we declared a relation with an inverse field in the schema. We'll put this in a function called makeDoors. Note how makeDoors is parameteric with respect to the semantics of the door. It receives an initialized manager and uses that manager to create the door. The DataManager is able to create a door regardless of the mixins that have been added. It thus only relies on data structure, and does not care about semantic extensions that have been given to the DataManager.

```
function makeDoors(manager) {
    let doors = new manager.StateMachine({"name": "doors"})

    let opened = new manager.State({"name": "opened"})
    let closed = new manager.State({"name": "closed"})

    doors.start = closed;
    doors.states = [opened, closed]

    let transOpen = new manager.Transition({"from": closed, "to":
        opened, "events": ["open"]})
    let transClose = new manager.Transition({"from": opened, "to":
        closed, "events": ["close"]})

    return doors
}
```

Figure 22:

Just like makeDoors, we can define a parametric function called *execute* that takes a state-machine and a sequence of events and returns a list of all transitions that were executed, and optionally an error if something went wrong. We terminate execution when an error is thrown or when no transition can be found for the event. Figure **??** illustrates how the door state-machine can be used with Logging and Locking, as well as a demonstration of possible errors when an incorrect sequence of events is given, or when the object is locked.

```
function execute(machine, events) {
    path = [machine.start]
    for(let event of events) {
        for(let transition in machine.start.transitions_out) {
            if(event in transition.events) {
                try {
                    machine.start = transition.to
                    path.push(transition.to)
                } catch (e) {
                    path.push("error: " + e)
                    return path
                }
            }
        }
        if(path.size -2 != events.indexof(event)) {
            path.push("error: there is no outgoing transition: " +
                event)
            return path
        }
    }
    return path
}
```

Figure 23:

We can now simply run the doors state-machine by invoking makeDoors, and then call execute with a list of events. Figure 24 illustrates the output that is produced by *execute* for three sequences of events. The first is succesful, while the second one contains an invalid sequence. For the third, we first lock the door before invoking execute and as a result, the door will throw an error that it is locked.

```
let manager = new DataManager(machineSchema, Logging, Locking)

let door = makeDoors(manager)
let events = ["open", "close", "open"]
let result = execute(door, events)
// result = [closed, opened, closed, opened]

events = ["close", "close", "open"]
let success, result = execute(door, events)
// result = [opened, closed, "error: there is no outgoing
    transition close"]

door.lock()
let success, result = execute(door, ["open"])
// result = ["error: the StateMachine is locked!"]
```

Figure 24:

7 Case Study: Performance of the MD4JS state machine against a regular ES6 class-based implementation

We'd like to know how Managed Data in JavaScript performs when compared to a regular ES6 class-based implementation. For this approach we'll define a semantically equivalent program that implements both logging and immutability, in addition to the type-checking that is required in MD4JS. The *execute* function should be equal to that of the managed implementation, with the exception of read/write operations using the dot operator. These must be replaced with method invocation in order to ensure that the code that implements the cross-cutting concerns is executed.

The evaluation will be done in two ways: We'll compare both the amount of lines of code written, as well as the execution time for both programs. We'll take 15 samples of both programs, average their execution time and then compare the results. For these two applications, we'll define a set of constraints which will guide us to nearly identical implementations:

- State-machine creation and execution logic should be equal in both implementations
- Both StateMachines must accept a loglevel which is oneOf ["debug", "info", "exception", "none"].

- When an object is initialized, a message should be logged with the initial values for this object
- When a property is modified, this should be logged
- When an exception occurs for a property, this should be logged
- Each object should provide a lock() and unlock() method, implementing immutability
- When an object is locked, it should throw an exception when an attempt is made to modify its properties. For a regular we'll limit this to it's methods that have side-effects on its data
- Timing should cover the entire state machine, from before creation until after execution to ensure that all aspects of the implementation are taken into account.
- Both applications should be transpiled to ES5 using the babel transpiler in order to eliminate performance difference due to transpilation, since babel has some known flaws where a piece of transpiled code is up to 50 times slower than a regular JS implementation.

7.1 Building the regular state machine

Just like the three classes defined in the *machineSchema*, we create three classes: a StateMachine, a State and a Transition. Properties are added in the constructor, and we define getters and setters for each. Next, we add locking behaviour. Each class receives a *this.locked* property, that can be changed by invoking the *lock* and *unlock* methods. In each method with side effects (data mutations), we check if the object is locked before making a change. This is the first part where the advantage of MD begins to appear: we must perform this check in multiple places, as opposed to the single mixin that was defined in the managed approach. The same holds for type checking, before making a change the given value must be validated. This type information is written in the methods itself. For larger code bases, this would turn into a problem because each property can be validated in multiple places. Therefore, if we make a change to the type, we must visit each of these places and refactor existing code.

Finally, we must add Logging behaviour. Since we added the requirement to accept a log level, the managed approach must also be extended to implement this. It has not been mentioned so far, but the final implementation for the DataManager also offers the possibility to give additional initialization info to mixins, both at the class level, as well as for individual instances. We can use these keyword arguments to initialize the StateMachine with a different log level. And in our Logging implementation we must do the same with the class's constructors. Logging initialization and message printing can be generalized into a class that the StateMachine, State and Transition can extend. We can then add logging invocations to these classes where needed. The resulting code

	class-based	**managed data**
minimum execution-time(ms)	17.851	205.280
maximum execution-time(ms)	20.944	219.162
average execution-time(ms)	18.783	211.233

Table 1: Performance of a class-based and managed door (in milliseconds)

	speedup
minimum speedup	8.1%
maximum speedup	10.2 %
average speedup	8.9%

Table 2: Performance loss of the managed implementation of a door as a percentage of the class-based approach

can be seen in Appendix A. The extension for the Logging mixin can be seen in Appendix B.

7.2 performance

In order to measure execution time we'll use JavaScript's *console.time* and *console.timeEnd* methods which we'll invoke at the beginning and end of both state machine implementations. We'll run this experiment 15 times and take an average, minimum and maximum of the measurements. Table 1 shows the minimum, maximum and average execution times for both the managed and the class-based implementation. These results have been obtained by executing a mix of successful and failing sequences of events, as well as a very very large sequence of successful events. Table 2 contains the minimum, maximum and average speedup as a percentage of the class-based implementation. A speedup of 10% means that the managed implementation is 10 times slower than the regular implementation.

7.3 Lines of code

The lines of code comparison can be divided into three distinct sections for Managed Data, and two sections for the regular implementation. For MD we have 1) the structural definition of data (the schema), 2) the cross-cutting concern (CCC) implementations (mixins) and 3) controlling logic (the part that deals with creation and execution of the state machine). The regular implementation combines 1. and 2. into a single class definition. As defined in the constraints above, the controlling logic part is equal to that of the Managed implementation. We'll use the SLOC (Source Lines of Code [10]) measure for the total amount of code, as well as structural definition and concern implementation separately. In Table 3 we observe that the managed program requires 22.8% fewer lines of code than the regular implementation. When we omit schema definition from this comparision, this reduction rises to 33.7%.

Table 3: Lines of code of two state machine implementations

	class-based	**managed data**	**reduction (%)**
total LOC	307	237	22.8%
LOC without schema	208	138	33.7%

8 Case Study: MD4JS for single page web applications

Besides performance, we'd also like to evaluate MD4JS on the aspect of practical applicability, and what better way is there to evaluate a JavaScript application than by building a web-application? So we're going to build a small web-framework according to the model-view-controller pattern (MVC)[11]. Recall that Managed Data is a model driven approach, so our goal is to use the MD framework that we've build earlier as the Model layer without making any changes to it. We're going to build a Controller and a View layer on top of this. Our data will be rendered in a simple HTML page that we'll make interactive using JQuery.

We'll reuse the state machine, and we'll transform it into an interactive web page. We'll start by defining the view; a collection of parametric methods that take a managed state machine and use it to render its output as HTML to a template. When the template is rendered, the view adds listeners to buttons, and maps them to a corresponding method on the controller class.

The controller defines methods that correspond to the buttons on the page. When a button is pressed, the corresponding action will be invoked on the controller, which will then use the *execute* method from figure 23 to process input. In this case we use an HTML text-field that accepts a list of comma separated events (strings). The view provides a method that is able to retrieve the value of this field, so that the controller can access it.

8.1 base controller

The basic controller, from which all further implementations will inherit, must at least do the following:

- Respond to actions that are invoked on the HTML page. It should be possible to invoke an action through all Events defined by JavaScript, such as onClick, onMouseOver, onMouseOut etc. (see JS Event documentation)
- Modify the model based on the action that was invoked. For this, actions will simply use the methods defined in the Managed Object itself, as well as the mixins that it implements.
- Notify the view when an object changes, such that it can re-render the new data. This results in the ability to develop smooth single-page applications

The base controller implementation is actually very simple: its constructor receives an initial value for the model, which is an instance of the schema's main Klass, a view object and a Data Manager which it can use to create new model instances. At the end of the constructor, the controller passes itself to the view, such that we can make callbacks to this controller from the view. Besides the constructor, we provide two template callbacks (empty methods) that can be filled in by extending Controller classes: *viewLoaded* and *afterUnload*. The first is used to implement any initialization logic that should be executed once the view finished rendering the initial template. The latter is used for cleaning up any residual data, or executing callbacks before a new page (and thus a different controller) is loaded.

8.2 base view

Just like the base controller, the base view consists of a very minimal set of initialization logic that each view must possess. The constructor receives a template name which should be rendered, as well as a JQuery representation of an HTML element in which the template should be rendered. Next, we append an empty $< div >$, load the template from a file, and render the HTML in the newly created div. As descibed above, we need a *setController* method to establish a two-way link between view and controller. We'll also add a *load* and *unload* method which respectively shows and hides the render element. These methods also invoke the corresponding callback on the controller. Finally, we add an *init* method. This init routine uses JQuery to search the template for HTML elements with the **.action** class. It then parses the *event* property of that HTML element and uses this to add a corresponding event listener that invokes an equally named method on the controller. This allows us to define custom actions and their corresponding handlers on any controller that extends the base controller.

In figure 10 we discussed how to build a schema that defines a state machine. The next step is to create a template, a view and a controller. We add a single input field with id *"events"*, one button with an *event* property with value "execute". Adding this event will result in the invocation of an equally named method whenever the button is clicked. We'll also add four elements where we can render the machines output:

- *machine-state* - this is where the current state is displayed.
- *machine-locked* - this holds a boolean value that indicates whether the machine is locked
- *machine-execution* - this displays any logging output from the logging mixin
- *machine-view* - this element renders a textual representation of the machine, including states and transitions

We'll now define the doors controller. We'll start by adding the *makeDoors* method from section 4. Upon initialization the controller should also remember the original starting state, initialize an array in which log output can be stored and add a counter for the number of errors that occurred during execution of the state machine. In the *viewLoaded* callback we'll invoke three render methods on the view that we will discuss later: *renderInfo*, *renderCurrentState* and *renderMachine*

Next, we'll implement the *execute* handler for the button that we added to our template. For this, we'll reuse the execute method that we used to evaluate the performance of the state machine in section 4. We'll need to make some small modifications. We need to rewrite it such at it operates on the *this.model* field, rather than an object that is given as an argument. We'll also need to add calls to the appropriate render functions after execution.

Since we want to display this state machine model with various mixins, we should map this specific model to the corresponding view and controller. This is essentially a very basic form of routing where a click event on a button corresponds with one application. It is possible to expand this to a type of routing called hash-routing, where a URL is postfixed with a hash(#), followed by a path. This splits the URL into two sections. The part before the hash will be used by the backend server, while the frontend will interpret anything following the hash. However, since the purpose of this section is to implement examples using Managed Data, and not to build a fully functional web-framework, we'll skip this step and keep the simple button-application mapping. Based on the button that was clicked we'll pass a string argument to the initialization function, and using a switch-statement we can initialize the corresponding objects like such:

```
1  function initDoors(type) {
2      switch(type) {
3          case "loggingDoors": {
4              manager = new DataManager(machine_schema, Logging)
5          view = new DoorsView("stateMachine/doors.html",
               view_element)
6          controller = new DoorsController(view, manager)
7
8          }
9          case "loggingLockingDoors": {
10             manager  = new DataManager(machine_schema, Logging,
                   Locking)
11             view = new LogingLockingDoorsView("stateMachine/
                   loggingLockingDoors.html", view_element)
12             controller = new DoorsController(view, manager)
13         }
14     }
15 }
```

Figure 25:

8.3 Transforming the doors statemachine into a regular expression validator

We can implement a very basic regular expression validator for gmail addresses by extending the doors controller such that it will view each submitted character in a string as a transition. We can use the state-machine schema as is, and we'll use it to create a machine that represents a regular expression for a gmail address. A state machine that implements a regular expression is valid if and only if each character in the string results in a successful transition. There should be no characters left in the string, and the machine should be in one of it's accepting states. We can thus extend the doors *execute* method by checking whether no errors occurred and if we have indeed arrived in the final state. The source for this implementation, among other examples, can be found in the source on github.

9 Evaluation

Looking back on the project we can conclude that at a reduction of performance, we can achieve a proper separation between data definition and behaviour. JSON schema is a straight forward way of describing models which has a positive effect on readability of the resulting program. Cross-cutting concerns only have to be implemented once since they are written to apply their logic on a single property. The DataManager then applies this logic to each property that's defined in the schema. This makes mixins easily reusable and results in a reduction of code that needs to be written when compared to regular ES6 programs. A reduction of 22.8% can already be achieved in a small program of 300 lines of code which implements three cross-cutting concerns. Because Managed Data is integrated with the host programming language, JavaScript, we can use the generated objects as if they were regular JavaScript objects. Because of this, controlling logic remains nearly identical to that of an ES6 class-based approach.

Implementing the web-application went just as expected: building the model layer took no effort at all, we could just plug in a DataManager and a schema and it worked. We then only have to tell a controller: here is your DataManager, when an action is invoked, these are the model's values that should be changed, and then we simply pass the model to the view. The view reads data as if it were any other JavaScript object and then interpolates this into an HTML template. Furthermore, it was really easy to reuse the mixins that we defined during the development of the state-machine and graph editor (two examples, see source code on github). We pass the mixins to a DataManager, and it works for any given Schema. This made it easy to reason about the implementations, since we only need to worry about when data should be modified and how it should be displayed. Questions like "should the data even be modified at all in this case?", "Does the given data have the correct type?" and "How should the data be modified?" were questions that were not relevant at all, since the

DataManager takes care of this.

Of course, not everything about Managed Data is better. We have quite a big loss of performance of about 91%, which means that MD would probably not be suitable for things like servers, games or other programs that handle large amounts of data. However, this performance loss wouldn't be that noticable in applications like a JavaScript webpage, where data only needs to be processed when the user interacts with an element, or when a change is pushed to the client. Here, the amount of data that is being sent tends to be small and infrequent and nearly all modern devices, including smart-phones should be able to handle this.

During the webapp case study, it proved to be difficult to handle serialization/deserialization of MObjects for persistence. This was due to the choice to save and load multiple MObjects with a single save invocation. This added the requirement to recursively save MObjects, which didn't go well with cyclic references between the objects. A better approach would have been write a save method that is parametric w.r.t. data semantics just like the *makeDoors* function. The persistence mixin would then only handle saving a single object, while the parametric save function decides which objects should be persisted.

10 Future and related work

There also is a Java implementation of Managed Data which mostly works in the same way[3]. However, since Java does not support mixins, the addition of behaviour is done through interfaces. Managed data has also successfully been implemented in ruby [4]. Python would be a language that also lends itself well to managed data, since it offers many useful tools like Meta classes, mixins and a built-in way of intercepting method calls and attribute access by overriding the __getattr__ and __call__ methods.

The Java Spring framework takes an Aspect Oriented approach to separating cross-cutting concerns, by dynamically extending method calls if a certain condition is met, such as *if method name contains "set" then log the affected property and value.*

A future extension of Managed Data could be to extend it such that it supports the GraphQL[12] JSON schema's. These describe a GraphQL API endpoint and can be generated by the back-end. This could then be plugged into Managed Data directly. When combined with an API synchronization mixin, we could then use this to completely generate the models needed for the corresponding front-end application. From there on, a programmer only has to be concerned with the way data is presented to users and handling actions given by users.

Appendices

Appendix A: Regular JavaScript implementation for a state machine that implements the logging and immutability cross-cutting concern

Appendix A1: Doors Machine (JS): Class that can be extended to implement basic logging

```
let loglevel = "info"

class Logger {
    constructor() {
        let available_loglevels = {
            "debug": 0,
            "info": 1,
            "exception": 2,
            "none": 3
        }
        if(loglevel in available_loglevels) {
            this.loglevel = available_loglevels[loglevel]
        } else {
            this.loglevel = available_loglevels["exception"]
        }
    }

    logPropertyChange(property, event, value) {
        if(this.loglevel < 2) {
            console.log(`event '${event}' on property '${property}'
                 of <${this.constructor.name}> with value`, String(
                value))
        }
    }

    logInit(initial_values) {
        if(this.loglevel < 2 ) {
            console.log(`initializing object of type <${this.
                constructor.name}>`)
            if(this.loglevel == 0) {
                console.log(initial_values)
            }
        }
    }

    logException(property, error) {
        if(this.loglevel < 3) {
            console.log(`exception occurred for property '${
                property}' in <${this.constructor.name}>: "`, error
                .message, '"')
            console.log(error.stack)
        }
        throw error
    }
}
```

Appendix A2: Doors Machine (JS): StateMachine implementation

```
class StateMachine extends Logger {
```

```
    constructor(name) {
        super()
        if(!name instanceof String) {
            let e = new TypeError("name bust be a string")
            this.logException("name", e)
        }

        this.name = name
        this.states = []
        this.locked = false

        this.logInit({
            name: this.name,
            states: this.states,
            start: this.start,
            locked: this.locked
        })
    }

    lock() {
        this.locked = true
    }

    unlock() {
        this.locked = false
    }

    setStartState(start) {
        if(this.locked) {
            let e = Error("the machine is locked")
            this.logException("start", e)
        }

        if(!start instanceof State) {
            let e = TypeError("start must be an instance of State")
            this.logException("start", e)
        }

        this.logPropertyChange("start", "set", start)
        this.start = start
    }

    addState(state) {
        if(this.locked) {
            let e = Error("the machine is locked")
            this.logException("states", e)
        }

        if(!state instanceof State) {
            let e = TypeError("state must be an instance of State")
            this.logException("states", e)
        }

        state.setMachine(this)
        this.logPropertyChange("states", "push", state)
        this.states.push(state)
    }
```

```
59
60      toString() {
61          return "<StateMachine>"
62      }
63  }
```

Appendix A3: Doors Machine (JS): State implementation

```
1   class State extends Logger {
2       constructor(name) {
3           super()
4           if(!name instanceof String) {
5               let e = TypeError("name must be a string")
6               this.logException("name", e)
7           }
8
9           this.name = name
10          this.transitions_in = []
11          this.transitions_out = []
12          this.locked = false
13
14          this.logInit({
15              name: this.name,
16              transitions_in: this.transitions_in,
17              transitions_out: this.transitions_out,
18              locked: this.locked
19          })
20      }
21
22      lock() {
23          this.locked = true
24      }
25
26      unlock() {
27          this.locked = false
28      }
29
30      setMachine(machine) {
31          if(this.locked) {
32              let e = Error("the state is locked")
33              this.logException("machine", e)
34          }
35
36          if(!machine instanceof StateMachine) {
37              let e = new TypeError("machine must be an instance of
                    StateMachine")
38              this.logException("machine", e)
39          }
40
41          this.logPropertyChange("machine", "set", machine)
42          this.machine = machine
43      }
44
45      addTransitionIn(transition) {
46          if(this.locked) {
47              let e = Error("the state is locked")
48              this.logException("transitions_in", e)
49          }
```

```
50
51          if(!transition instanceof Transition) {
52              let e = new TypeError("transition bust be an instance
                    of Transition")
53              this.logException("transitions_in", e)
54          }
55
56          transition.setTo(this)
57          this.logPropertyChange("transitions_in", "push", transition
                )
58          this.transitions_in.push(transition)
59      }
60
61      addTransitionOut(transition) {
62          if(this.locked) {
63              let e = Error("the state is locked")
64              this.logException("transitions_out", e)
65          }
66
67          if(!transition instanceof Transition) {
68              let e = new TypeError("transition must be an instance
                    of Transition")
69              this.logException("transitions_out", e)
70          }
71
72          transition.setFrom(this)
73          this.logPropertyChange("transitions_out", "push",
                transition)
74          this.transitions_out.push(transition)
75      }
76
77      toString() {
78          return "<State>"
79      }
80  }
```

Appendix A4: Doors Machine (JS): Transition implementation

```
1   class Transition extends Logger {
2       constructor(events) {
3           super()
4           if(!events instanceof Array) {
5               let e = new TypeError("events must be an instance of
                    Array")
6               this.logException("events", e)
7           }
8
9           for(let event in events) {
10              if(!event instanceof String) {
11                  let e = new TypeError("events must be an array of
                        strings")
12                  this.logException("transitions_out", e)
13              }
14          }
15
16          this.events = events
17          this.locked = false
18
```

```
        this.logInit({
            events: this.events,
            locked: this.locked
        })
    }

    lock() {
        this.locked = true
    }

    unlock() {
        this.locked = false
    }

    setFrom(state) {
        if(this.locked) {
            let e = Error("the transition is locked")
            this.logException("from", e)
        }

        if(!state instanceof State) {
            let e = new TypeError("state must be an instance of
                State")
            this.logException("from", e)
        }

        this.logPropertyChange("from", "set", state)
        this.from = state
    }

    setTo(state) {
        if(this.locked) {
            let e = Error("the transition is locked")
            this.logException("to", e)
        }

        if(!state instanceof State) {
            let e = new TypeError("state must be an instance of
                State")
            this.logException("to", e)
        }

        this.logPropertyChange("to", "set", state)
        this.to = state
    }

    toString() {
        return "<Transition>"
    }
}
```

Appendix A5: Doors Machine (JS): parameteric *makeDoors* and *execute* methods

```
function makeDoors() {
    let doors = new StateMachine("doors")

    let stateOpened = new State("opened")
```

```
5       let stateClosed = new State("closed")
6       let stateLocked = new State("locked")
7
8       doors.setStartState(stateClosed)
9       doors.addState(stateClosed)
10      doors.addState(stateOpened)
11      doors.addState(stateLocked)
12
13      let transOpen = new Transition(["open"])
14      let transClose = new Transition(["close"])
15      let transLock = new Transition(["lock"])
16      let transUnlock = new Transition(["unlock"])
17
18      stateOpened.addTransitionIn(transOpen)
19      stateOpened.addTransitionOut(transClose)
20
21      stateClosed.addTransitionIn(transClose)
22      stateClosed.addTransitionIn(transUnlock)
23      stateClosed.addTransitionOut(transOpen)
24      stateClosed.addTransitionOut(transLock)
25
26      stateLocked.addTransitionIn(transLock)
27      stateLocked.addTransitionOut(transUnlock)
28
29      return doors
30  }
31
32  function execute(machine, events) {
33      let errorCnt = 0
34      let executionLog = []
35
36      for(let event of events) {
37          success = false
38          for(transition in machine.start.transitions_out) {
39              if(event in transition.events) {
40                  try {
41                      machine.setStartState(transition.to)
42                      executionLog.push("Executed transition from "+
                            transition.from.name+" to "+transition.to.
                            name+" with event "+event)
43                      success = true
44                  } catch (e) {
45                      errorCnt++
46                      executionLog.push("An exception occurred while
                            executing transition from "+transition.from
                            .name+" to "+transition.to.name+" with
                            event "+event)
47                  }
48              }
49          }
50          if(!succes) {
51              errorCnt++
52              executionLog.push("State "+transition.from.name+" has
                    no outgoing transitions with event "+event)
53          }
54          return {
55              errorCnt,
```

```
            executionLog: executionLog.join("\n")
        }
    }
}
```

Appendix A6: Doors Machine (JS): main function that runs some tests

```
export function executeStm(logLevel, longSequence) {
    loglevel = logLevel
    let doors = makeDoors()

    /* Run a successful sequence of events */
    let events = ["open", "close", "lock", "unlock", "open"]
    let initialStart = doors.start

    /* executes sucesfully */
    let result = execute(doors, events)

    /* Run a failing sequence of events */
    doors.setStartState(initialStart)
    events = ["open", "close", "close", "open", "close", "lock"]

    /* returns execution failure as result */
    result = execute(doors, events)

    /* Run some events, then lock the machine */
    doors.setStartState(initialStart)
    events = ["open", "close", "lock"]
    result_before_lock = execute(doors, events)
    doors.lock()
    /* throws error: machine is locked */
    let result_after_lock = execute(doors, ["unlock"])
    }
}
```

References

[1] M. Bruntink, A. van Deursen, T. Tourwe and R. van Engelen. *An evaluation of clone detection techniques for crosscutting concerns.* 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., Chicago, IL, 2004, (pages 200-209).

[2] *Source Code on github* https://github.com/rvanbuijtenen/ManagedDataJS/

[3] Theologos Zacharapolous, Pablo Inostroza, and Tijs van der Storm (ACM SIGPLAN 2016) - *Extensible Modeling with Managed Data in Java.*

Retrieved from *https://homepages.cwi.nl/ storm/publications/md4j.pdf.*

[4] Alex Loh, Tijs van der Storm, and William R. Cook. *Managed Data: Modular Strategies for Data Abstraction* (Ensō Paper 1 of 6). In: Onward!, 2012.

[5] Grover D., Kunduru H.P. (2017) *Meta Programming.* In: ES6 for Humans. Apress, Berkeley, CA (pages 117-127)

[6] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, Domagoj Vrgoč *Foundations of JSON Schema.* In: WWW'16 (pages 263-273), 2016

[7] Michael Droettboom, Space Telescope Science Institute *Understanding JSON schema*, retrieved from *https://spacetelescope.github.io/understanding-json-schema/reference/*

[8] Allan Ebdrup. *JSON Schema benchmark*, an open source benchmark tool for JSON schema validators. Retrieved from *https://github.com/ebdrup/json-schema-benchmark*

[9] *SLOC measuring tool for NPM* https://www.npmjs.com/package/sloc

[10] I. Heitlager and T. Kuipers and J. Visser. *A Practical Model for Measuring Maintainability.* In: QUATIC 2007 (pages 30-39)

[11] Nilesh Jain, Priyanka Mangal and Deepak Mehta (2014). *AngularJS: A Modern MVC Framework in JavaScript.* In: Journal of Global Research in Computer Science http://www.laputan.org/pub/papers/POSA-MVC.pdf

[12] Olaf Hartig, Jorge Pérez (2017). *An Initial Analysis of Facebook's GraphQL Language.* In: Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web. Juan Reutter, Divesh Srivastava, Juan Reutter, Divesh Srivastava , 2017, Vol. 1912, article id 11