# Final Report

## Streaming JSON Parsing

T. M. Booij
R. A. Klavers

Delft University of Technology

TU Delft Delft University of Technology

Challenge the future

# Final Report

## Streaming JSON Parsing

by

**T. M. Booij**
**R. A. Klavers**

**Bachelor of Science**

Faculty of Electical Engineering, Mathematics and Computer Science

Delft University of Technology

| | | |
|---|---|---|
| Supervisors: | Dr. ir. A. J. H. Hidders | TU Delft |
| | M. Wijngaard | Moxio |
| Bachelor Project Coordinator: | Dr. ir. Martha Larson | TU Delft |

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft Delft University of Technology

moxio

# Contents

# PREFACE

In front of you is the result of the Bachelor Project carried out by Tim Booij and Robert Klavers at Delft University of Technology as a conclusion of our Bachelor of Science in Computer Science and Engineering. Over the course of three months we researched possibilities of implementing a high performance system to parse and access large JSON datasets. This project was conducted at Moxio B.V.

We would like to thank the people at Moxio B.V. for supporting this project. Especially Merijn Wijngaard who provided constant feedback and counseling when needed and H. van der Kolk for facilitating this project.

Another thanks goes to assistant professor J. Hidders, part of the Web Information Systems group at Delft University of Technology, our supervisor. His enthusiasm and guidance really helped us throughout the project.

*T. M. Booij*
*R. A. Klavers*
*Delft, July 2015*

# 1

# INTRODUCTION

*Moxio B.V.* is a company that busies itself a lot with building web applications to facilitate data and information flow from different parties involved with a large construction project, e.g. *Spoorzone Delft*, *Rotterdam Centraal*, etc. *Moxio* is focused on supporting these data-transfers and validating this data.

There is an ever increasing amount of data being sent between various application in use at *Moxio*, as they increase the amount of data being validate so does the size of the resulting data sets. For example, *Moxio* validates technical AutoCAD drawing to analyze if they conform to current project rulesets. The amount of data they retrieve from these drawings increases with the variety of specifications they can validate. Data retrieved from a large drawing can be in the range of hundreds of MB. Therein contained are properties like line length, thickness and colour, but also metadata concerning the type of drawing, geographic area, scale, etc. At some point it's no longer feasible to parse all this data at once and store it in memory. The need arose to find a solution to parse and access this data that has a small memory footprint to be able to handle this increasing amount of data.

This report describes the process of coming up with a solution for this problem and implementing it in the context of *Moxio*. In chapter 2 we begin by describing and detailing which approach we took to the problem, which is based on our research phase, described in detail in appendix C. In chapter 3 we illustrate how we approached the implementation, issues we ran into during this phase and why we made certain decisions. The next chapter, chapter 4, deals with the various ways we used to make sure our solution was working, robust and of high quality. Chapter 5 reflects on the project as a whole, we evaluate the way things have gone and how we would improve certain aspects.

# 2

# DESIGN

## 2.1. INTRODUCTION

This chapter is a direct result of the research phase described in appendix C. Due to the way the project is organized and the exploratory nature of the problem, we had a lot of freedom to really dive into the theory of parsing and other related techniques. This research report gives a rundown of the requirements of the project and provided us with the necessary knowledge to devise a solution. Based on this research we drafted an approach using concepts from various sources and creating a method applicable in the context of *Moxio B.V.*.

Based on our research we opted to use an approach using inspiration from various existing solutions. On one end the parser needs to be streaming and able to parse from a given index in the data. On the other end the objects providing access to the data, i.e. those implementing ArrayAccess and thus being exposed to the user, do not need to directly store their values. They will store the index and length to be able to determine where the value for a given key can be found.

This chapter will discuss this design and provide a high level overview of our system. It will discuss a number of design decisions we faced and why we made certain decisions. First an overview of the various components is given, then we show how these components will interact and some caveats concerned with this.

In section 2.2 we will define a number of components and their relation to each other. In the following sections we expand upon these components. The parser, which lies at the heart of the system, is discussed in section 2.2.2. The Index overlay, which is responsible for the interaction between the client and parser data, is described in section 2.2.3.

## 2.2. SYSTEM OVERVIEW

In section C.5 of our research report we outlined our vision of the direction we think to head in. This section aims to further specify and formalize this approach. We do this by introducing the interaction between various component and explain their inner workings as we envision them.

### 2.2.1. COMPONENT INTERACTION

To get an idea of the interaction between components we give a simple overview of the process from creating a parser to requesting the data, before explaining the various components in more detail.

At first the parser has to be initialized. It is created on top of a string, file or any other seekable format. This is wrapped by a standard datastream format. An initial 'collection' object has to be created. The Collection object is responsible for providing an API to the client by implementing ArrayAccess and IteratorAggregate, two PHP interfaces to allow the client to use standard PHP constructs. Instead of having this collection interact

directly with the parser an additional construct is necessary. This has functionality to retrieve properties and iterate over properties.

From this overview we identified a number of major components vital to the system. These components are covered in more detail in the following sections.

### 2.2.2. JSON PARSER

At the core of our system lies the parser, it is responsible for converting a stream of 'characters' to a sequence of JSON elements. As we saw in section C.3 of the research report there are multiple ways to implement a parser. Important for us is the ability to constantly request a next element. To allow the parser to determine the type of the next element you need to keep track of state or the current 'context'. This means that the parser must know what the previous elements are to be able to determine the next element. An example:

The parser needs to parse the following piece of JSON: `{"a": "b"}` This will result in the following sequence: *"Object Start", "Property name", "String value", "Object end"* To be able to determine that `"b"` is a *String Value*, whereas `"a"` is a *Property Name* the parser has to remember state. A stack would allow this, once it encounters a `{` it will, for example push a state `OBJECT` and `PROP_NAME` onto the stack. If it then moves to the next character it will pop the state and see that this string should be a property name. Similarly, after a `:` the parser will expect a value (which could be another object).

With this design we aim to be able to validate the input. This allows us to provide descriptive error messages to be able to determine where there is an error in your JSON. As state is only of importance within an array or an object, it will also allow us to start at any point in the data (provided it starts with either an '`{`' or '`[`') and gives us the ability to parse from there. In our research report we showed a few diagrams that describe the possible states and transitions C.3.2.

### 2.2.3. INDEX OVERLAY

There are various ways to create and categorize parsers. In the context of the project, memory-efficiency is one of the key points, which is why different access methods should be considered. When data is sequentially accessed, the data processor can only access the data which is currently being looked at. This parser, unfortunately, does not have access to data found earlier, nor can it look at data which lies ahead. Sequential access is very vast, which is one of the requirements for our system. However, accessing data in this manner, limits the system to the window which has just been parsed.

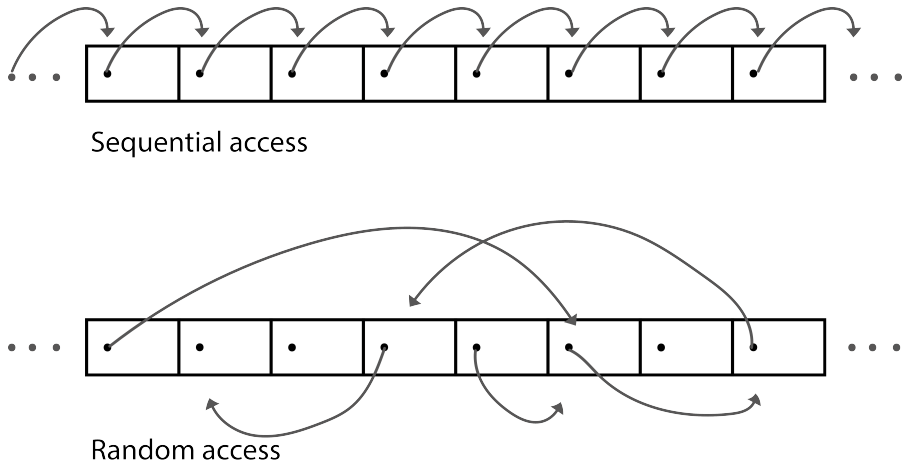


Figure 2.1: Sequential and random access

One of the prerequisites of the system was that it should have random access. Using this way of data access, gives the system the opportunity to move back and forth over the data at hand. Random access is not necessary slower than sequential access, but it does cost more memory. This is because it makes it possible to access data which has already been parsed and has to save a lot more data. We will elaborate this in

section 2.2.3. More information on this can be found in our research report, appendix C.

The initial idea for the system was to implement random access, which would be one of the key points. Creating an extra layer with indices pointing to the data at hand would give the system a handle to efficiently search for data. This also makes it possible to skip large amounts of data which are not needed to remember. A correct implementation would vastly reduce the memory usage. Figure 2.2 depicts this idea of creating an extra map with indices.

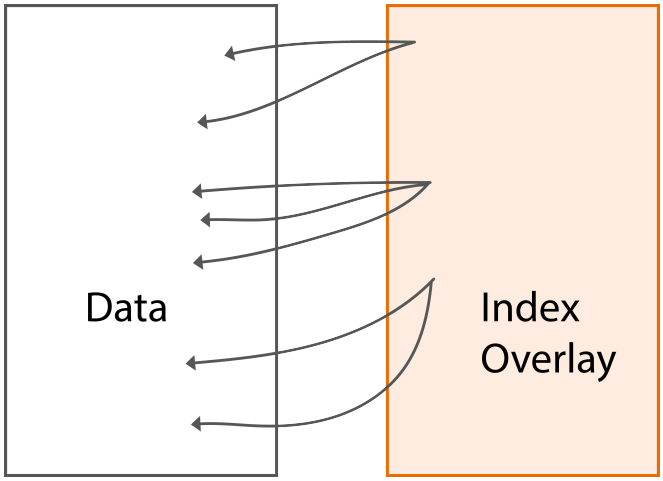


Figure 2.2: Idea to map values and indices

This idea is based on a design called VTD for XML. VTD stands for Virtual Token Descriptor. Our implementation would create a pointer for each property name found in the JSON data and attach information about its value, like the length, the starting index and its children (if it has any). Implementing this would thus create a whole new dataset with only key and value pairs. In this overlay it would be possible to easily search for and retrieve values given the property name. This could be implemented with simple nested arrays and would look similar to a tree-structure. Implementing this would look like something in listing 1

```
temp[this->getElementAtIndex(index) =
    array(
        "type" => this->parsedTokens[index + 1]->type,
        "startIndex" => this->parsedTokens[index + 1]->startIndex,
        "length" => this->parsedTokens[index + 1]->length,
        "children" => array(...)
    );
```

Listing 1: Factory example

### LAZY LOADING

This implementation gives the system a nice basis for the initial idea. The next step would be to making it grasp the concept of lazy loading. This is a design pattern which can be used to defer the initialization of code, until it is actually needed. To create such a pattern in the design, the system can simply create the index overlay up until the point in which the right property name has been found. This would be easy to implement and a good solution in the situation in which we only request property names located at the start of the dataset. But when a situation occurs that the value of a property name is desired located at the end of the document, this would not be a good resolution. The index overlay would then be built up almost completely.

### MEMORY USAGE

With the focus on memory efficiency, the system should handle this in a different and smarter manner. Ideally, the overlay should only build the references which are already parsed. The rest of the objects around

it are not important and should be skipped. This creates a mapping which contains minimal information, but at the same time can also be used to quickly find other surrounding objects. Because this creates a tree structure, a neighbour can be found by following the tree until the point where it has to take a different object. The starting index of the parent could be used to quickly navigate to the right point in the tree.

At this point, an overlay can be created to quickly navigate in the provided dataset. Of course this extra overlay can grow big, when the dataset is also big and lots of requests are made. This would miss the point of keeping the memory usage low. By introducing a memory management technique of some sort could solve this. Something simple, like a limit on the amount of stored references would already prevent this from happening.

# 3

# IMPLEMENTATION

## 3.1. INTRODUCTION

Having specified the design of the system, the next step was implementing this design. This chapter will focus on the implementation process and decisions made during this stage. We will go through all major components and explain how we tackled certain issues and how our approach for these issues evolved throughout the project. Having a set of largely independent components allowed us to develop them side by side. We will go over the index overlay system in section 3.4.1, the parsing phase in section 3.2 and caching in section 3.3. This chapter will focus a lot on our code-patterns used. By using a couple of examples we will explain functionality of these components and their working.

### 3.1.1. ARCHITECTURE

To make sure that names of entities are clear and explicit components are consequently named and aggregated by functionality. Throughout development we settled on the following structure as seen in figure 3.1:

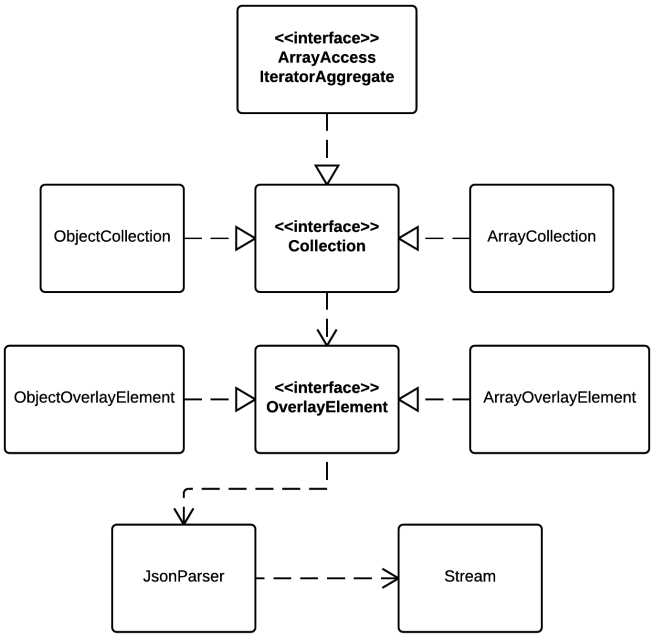


Figure 3.1: UML diagram of high level system overview

In other words, a `Collection` is used to implement functionality to use PHP's ArrayAccess and foreach constructs. Thus being able to use `$data["a"]`. An OverlayElement is used to communicate with the Parser to find and return a certain value for a given offset.

## 3.2. PARSER

The JSON Parser lies at the heart of the system. At its foundation it is nothing more than a state machine, rolling out a stream of Json Elements. As mentioned in section 2.2.2, we devised our own solution.

### 3.2.1. TOKENIZER

Initially we split up parsing in two phases, creating a token sequence and transforming this sequence into a sequence of JSON elements. The tokenizer recognizes the type of element and transforms it into a token. A token is nothing more than the type of element without any meaning. For example:

Table 3.1: Transforming characters to tokens

| `{` | `"a"` | `:` | `42` | `}` |
|---|---|---|---|---|
| Left curly bracket | String | Colon | Number | Right curly bracket |

The parser would then take these tokens and provide them with meaning:

Table 3.2: Parsing the tokens

| `{` | `"a"` | `:` | `42` | `}` |
|---|---|---|---|---|
| Object start | Property name | | Number value | Object end |

This allowed us to quickly construct this parser, and gave us the ability to develop the layers above quickly. Nevertheless this way of parsing eventually proved to be too slow and was developed without reflecting on the theory as laid out in section C.3.2 of our Research report.

### 3.2.2. MERGING TOKENIZING AND PARSING PHASE

Separating the tokenizing and parsing phase allowed us to reason about the workings of the parser more easily, it proved to be too inefficient. Therefore we set out to merge these two phases to allow them to occur simultaneously. At its core the general workings are similar, it still moves through encountered characters, skipping whitespace and providing context to the rest. But now we immediately try to infer the type of element at the moment it is encountered. This also allows us to define the parser more formerly. We more specifically defined the parser as a push down automaton keeping state in a stack as we reasoned in our research report.

#### STATE MACHINE

As mentioned before a pushdown automaton employs a stack to utilize more memory. For example, when the parser encounters an opening bracket, it will push this on the stack. When a matching closing bracket is encountered this will be popped off the stack. This way you are, for example, able to match opening and closing brackets, as demonstrated in section C.3.2 of the research report. But more importantly it allows you to give meaning to encountered elements. In the case of JSON, after an `{` is encountered, the state `Property Name` will get pushed on the stack as that is the next expected state. Every time a new element is encountered you will pop the top element of the stack, check if this is a valid state and then move on.

#### PERFORMANCE

Figure 3.2 shows the resulting increase in performance. This is a test on a size of 185MB containing data on city lots in San Fransisco. Each point is the time it took to parse 1000 rows of this file, to ensure that parsing time is independent of running time we plotted each of these points instead of taking overall running time. On average each iteration took 58% of the time it did before merging these two phases.

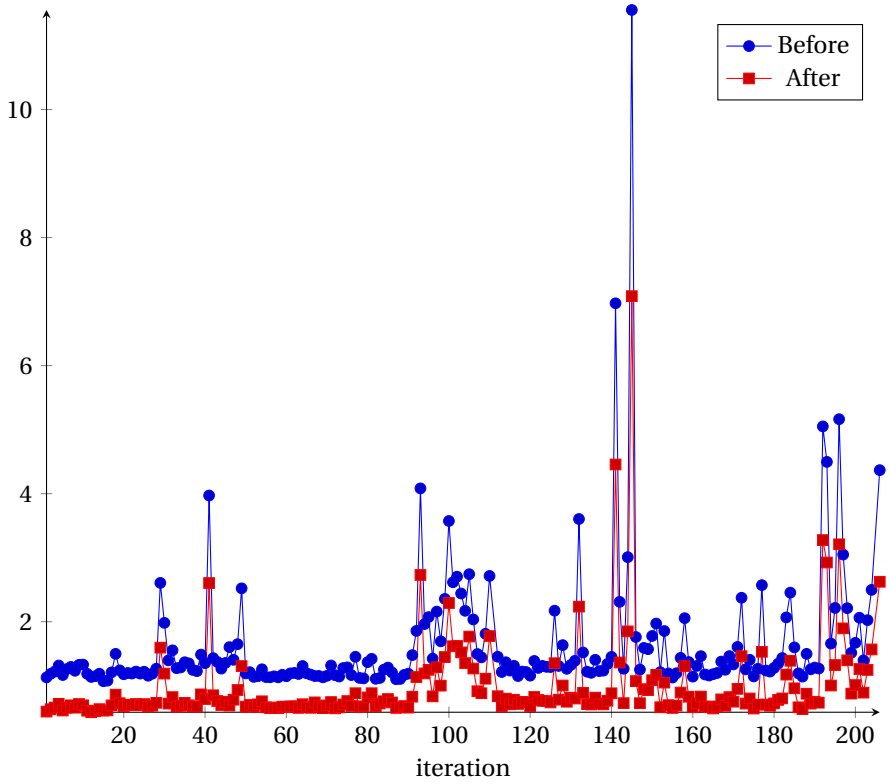


Figure 3.2: Refactor performance comparison

## 3.3. CACHING

By now, the system creates a datastream from the dataset and can correctly parse and retrieve information from a given JSON dataset. When comparing the memory usage at this point, it has significantly decreased compared more formerly to loading the original dataset as a whole into the memory. This is a great base for our system. The initial idea is now implemented as a system. The next step is to further improve performance and refactor code. An essential component still missing from the system is a way of saving the already parsed information. Now, each time the parser will discard all the data which has already been parsed. A way of caching has to be introduced in order to preserve just enough information to save a pointer to the information which the system has already parsed.

### 3.3.1. CACHING PROPERTY NAMES

Caching is a technique to speed up data lookups. Instead of directly reading the data from the stream, the data is first read from a cache if available. Accessing a cache is typically faster than accessing the data from its original stream. In our situation, the cache would contain a wrapper of information. If the cache would save all of the information, it would miss its point in the end. Looking back at the data source, which is a stream, it only needs an index to quickly seek to the information needed. This means only the reference to a certain point in the data have to be saved in the cache. As explained earlier, the JSON data works with property names and values, like this: `{"property_name": value}`. In theory, only the property name and the starting index of it have to be saved in the cache. With this information the system can check if a certain property name is already saved and directly jump to its starting index in the stream, which saves time.

Looking up data in a cache with starting points is a great technique to preserve a lot of memory, but first a more generic solution has to be implemented. Whilst forgetting about the starting index, the cache is left with a property name and its value. For now this will be sufficient, because the value at hand contains information about underlying objects. With these objects the system will be able to dive into the information and retrieve the next property name.

### 3.3.2. FIFO

Managing the size of a cache is very important, as the size of the input data could be too big to handle for the system's memory. This is why the system needs a technique which can manage how much data can be stored in the cache. A bunch of different caching techniques exist which would be applicable in the systems situation. Data could be retrieved from the cache on a time based eviction, on a least accessed base and so on. We have chosen the FIFO method, which means First In, First Out. This means, given the cache has reached its maximum limit of data and you wish to insert a new value, the earliest inserted value is removed to make space for the new value. Before the limit has been reached, every value can be inserted without anything being removed. We chose this solution, because the implementation is not very advanced and quickly gives great results.

### 3.3.3. PREFIX

Initially the system would take the property name parsed from the JSON, and store it with its starting index directly into the cache. So if we would want to retrieve the value stored in `data["menu"]["id"]`, we would create a cache which would contain `menu` and `id` respectively. However, JSON data can contain arrays, so arrays should be able to be accessed too. If we look at another attempt to retrieve information stored in `data["coordinates"][0][0]`, this would produce the wrong data. In this example, the value `0` would first be stored in the cache and then retrieved. So the system would return an object instead of the expected value. To solve this problem, we have extended the caching implementation to be able to deal with prefixing. In the first example, the value can be found by searching for `menu_id` and for the second example by searching for `coordinates_0_0`.

## 3.4. ARRAYACCESS

### 3.4.1. INDEXOVERLAY

As mentioned before the `IndexOverlay` provides access to the `JsonParser`. Its responsibility is to find and iterate over Json Elements. It has functionality to move through the data and find objects. Its interface has a few simple methods, similar to the Iterator design with a few subtle changes: `getElementAtCurrentOffset()`, `hasNextElement()`, `moveToFirstElement()`, `moveToNextElement()` and `moveToOffset(\$offset)`. The functionality of these methods should be self explanatory. Using such an overlay provides a useful abstraction of the parser and prevents having to write code manipulating the parse stream at the same place it's being accessed.

### 3.4.2. COLLECTION

Finally the classes implementing the `CollectionInterface` contain all logic to access an `IndexOverlayElement` into a PHP object that can be accessed through the standard PHP constructs. This means that this implements the PHP interfaces `ArrayAccess` and `IteratorAggregate` which dictate methods for traditional array manipulation, such as setting a value or reading a value. These methods will use the `OverlayElement` to find and retrieve values. The `IteratorAggregate` interface expects a `getIterator` method which returns an Iterator. This allows you to use a `foreach` construct to iterate over the elements by key and its respective value.

## 3.5. REFACTORING

During development, especially with a time limited project such as this, you will inevitably build up a certain technical debt. This means that due to "quick- and easy" features the code base will grow to be less maintainable. Therefore it is important to keep this in check and clean up and refactor regularly. This section will detail our efforts to implement proper patterns to ensure a clean codebase which is modular and maintainable.

### 3.5.1. DESIGN PATTERNS

Design patterns are solutions for common problems in software design. They provide well-proven and well-tested designs for certain problems. Here we describe a number of these patterns, what their goals are and how they helped us improve our code.

#### ITERATOR

As described in section 2.2.2 the iterator pattern was chosen to use for the Parser. The iterator pattern is used to traverse a sequence of elements. This is used to decouple the usage of the Parser from its implementation. And therefore allowing you to switch the Parser for a different one, as long as it returns JsonElements and implements Iterator. In our case the Parser provides a useful abstraction on top of the datastream.

#### DEPENDENCY INJECTION

Dependency injection is nothing more than taking an object and giving it its dependencies instead of having it instantiate them itself. For example an `IndexOverlayObjectElement` uses a `JsonParser` object, instead of creating an instance of `JsonParser`, it gets passed into its constructor. Using this pattern makes adherence to the single responsibility easier and allows for easier testing. Instead of having to find a way to mock the `JsonParser` instance the `IndexOVerlayObjectElement` has created, we can, for example, simply pass a `JsonParserMock` object which is an instance of `JsonParserInterface`.

#### FACTORY PATTERN

A factory pattern has responsibility of creating objects. In essence a factory is nothing more than an abstraction of object instantiation. Using factories to create objects encapsulates logic concerned with object creation, resulting in a clean call to instantiate an object. These factories can then also be injected where needed. For example, an `IndexOverlayObjectElement` needs to return a new instance if the requested value is another object or array. It can then use the factory to create this element. In listing 2 you see the method will return a new collection based on the type of the JSON element given, An array or object. The factory then also injects itself in the created `Collection` as this will make use of it to create new `Collection` instances.

```php
<?php
public function buildCollection($element) {
  switch ($element->type) {
    case JsonElementTypes::JSON_OBJECT_START:
      return new JsonObjectCollection(new IndexOverlayObjectElement($this->parser, $this->indexParser,
                $element->startIndex), $this);
      break;
    case JsonElementTypes::JSON_ARRAY_START:
      return new JsonArrayCollection(new IndexOverlayArrayElement($this->parser, $this->indexParser,
                $element->startIndex), $this);
      break;
  }
}
```

Listing 2: Factory example

Having these factories in place also allows us to move responsibility of creating new elements to a single place, which greatly improves modularity and testability.

#### DECORATOR

We wanted to be able to dynamically change and add functionality. Decorators help us do this by encapsulating added responsibilities and being able to add these to existing objects. For example, `JsonCollection` implements the `CollectionInterface` and provides methods to retrieve values from a JSON Element. The

```php
<?php
public function __construct(CollectionInterface $collection, CacheInterface $cache) {
  $this->collection = $collection;
  $this->cache = $cache;
}
```

Listing 3: CachingCollectionDecorator Constructor

`CachingCollectionDecorator` also implements this interface and requires an instance of a `JsonCollection` and an instance of `CacheInterface`. In listing 3 you can see an example how such a decorator is constructed.

This allows us to use the cache when we need to but use a normal collection otherwise. An example is given in listing 4

```php
<?php
  public function offsetExists($offset) {
  return $this->cache->has($offset) || $this->collection->offsetExists($offset);
}
```

Listing 4: Using the decorator

# 4

# Quality Assurance

## 4.1. Introduction

From the start of the project it was clear that the parser which was going to be built, had to be tested thoroughly. Our supervisors from *Moxio* and the *TU Delft* attended us that this would be very important. Test-Driven Development would be the most effective (and most simple) way to achieve this. Also the performance of the parser is, in our opinion, the most important piece of testing. Our whole idea was to efficiently parse the data, with minimal memory usage. This will be further explained in section 4.2. The quality of the final product was assured by different entities. These entities and the way they evaluated the working of the system and its code will be described in section 4.3. To conclude, in section 4.4 we will elaborate our choices for version control and continuous integration.

## 4.2. Testing

Testing is one of the most important aspects of programming, especially when developing library code. This section will dive into the effort put into testing the system.

### 4.2.1. Test Driven Development

The core code of our system is library code. There is a big difference between library code and application code, which need to be tested in a different manner. Application code is basically only used in one environment and can be changed to alter behaviour. Library code, however, is special, as it is meant to be reusable in different systems and applications, working under different circumstances. It should be easy to integrate or extend the code without any changes. Developing library code in our case means dealing with lots of edge cases in the specification. To work efficiently, we chose the software development process called Test-Driven Development (or TDD). TDD relies on short repetitive development cycles. Initially we created failing test cases, which pointed out a flaw or new functionality of the system with as little code as possible. After that we wrote or refactored code where necessary. This way of continuous testing worked very well and is also recognized and endorsed by SIG, which we will discuss in section 4.3.1.

### 4.2.2. Unit testing

To keep track of all of the components and to check whether all functionalities are still working after a new feature, we made use of automated testing. Unit testing is a great method of software testing by which individual units of source code are tested to decide whether they are good to go. Theoretically, a unit can be seen as the smallest possible testable part of the software. In our case we built up the complexity of testing, by first testing simple code and then kept increasing the difficulty when moving upwards in the class-chain. We used the PHPUnit framework to create our testing environment.

## 4.3. CODE QUALITY

To assure the quality and maintainability of the code, several measurements were taken to constantly improve it. Two measurements were conducted by the Software Improvement Group and code coverage and test volume could be tracked through a continuous integration server.

### 4.3.1. SIG

The Software Improvement Group (or SIG) is an independent code evaluation company, working together with the Delft University of Technology. They are specialized in providing fact-based insight into IT situations and give recommendations to improve. Our code was assessed twice by this company by consolidating low-level technical analysis on our system. The feedback provided from SIG was very valuable for us. Both times our system scored high on their maintainability-model. Most importantly, these two reflection points also gave us acknowledgement for our way of developing. In the second analysis, one important conclusion was that we kept writing test-code along with the production-code. This confirmed our software development process was successful, regardless of time-pressure. The feedback can be found in appendix D.

### 4.3.2. TEST VOLUME AND CODE COVERAGE

TDD is all about testing every small case which can be tested. During the development process, it is especially important that the procedure for writing production code stays the same. So for each improvement or new feature, failing tests are the starting points. Figure 4.1 depicts this very nicely. This figure shows the increase of tests is almost constant throughout the project. If we reflect on the feedback from SIG, we learn that whilst the volume of our production code doubled, the test code has also been doubled.

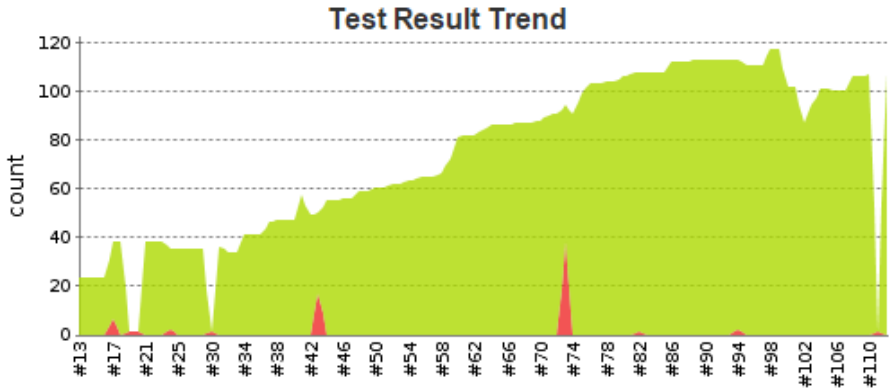


Figure 4.1: Code volume of our system July, 2015

Another important aspect of TDD is the amount of tested code. In theory every piece of code could be tested, resulting in 100% code coverage. However, in reality this is almost never the case, but most of the code should be covered. In figure 4.2 the code coverage of our system is almost 90%. This also confirms our development process, especially because the coverage is almost consistently above 80%.

## 4.4. VERSION CONTROL AND CONTINUOUS INTEGRATION

### 4.4.1. GIT AND GITHUB

For version control we chose to use Git and used GitHub as the web-based repository service. Git offers a few upsides towards other version control services, as for example SVN. Git is built with emphasis on support for distributed workflows and especially speed. Unlike other services, Git provides the user with a complete history and version-tracking capabilities. This enables the user to create branches for new features and easily merge these features into the master branch.

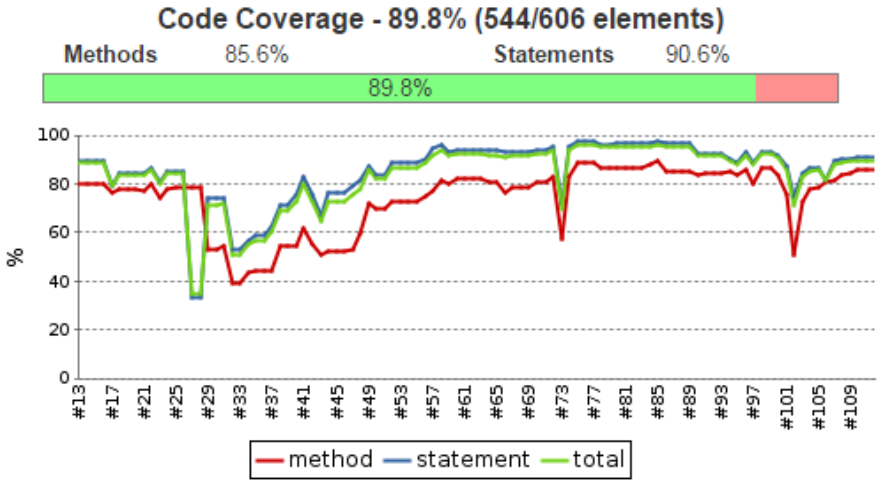


Figure 4.2: Code coverage of our system July, 2015

### 4.4.2. JENKINS

Jenkins is an application that manages job executions, for example building software projects. It can continuously build and test individual software projects. It can also monitor every job execution (internal or external). By using Jenkins we can run all tests after implementing each new feature or improvement to check whether all functionalities still work. *Moxio* uses Jenkins for continuous integration for each application they own. So for us it was an easy choice to create our project on *Moxio's* Jenkins server.

# 5

# PRODUCT EVALUATION AND RECOMMENDATIONS

## 5.1. FINAL PRODUCT

Throughout the project we have explored possibilities of high-performance JSON parsing in PHP. We have created solutions and discarded them. In the end we delivered a project which is able to relatively quickly parse large amounts of data, while maintaining a controllable memory footprint. It utilizes caching to increase performance. Nevertheless there are still a number features which we were unable to implement due to time constraints. A number of these features will be discussed in section 5.2.

We experienced that high performance random access to large JSON data is difficult to implement. You are forced to make trade-offs due to the paradoxical nature of these two approaches. Despite this, we believe that the product we created proves that this is worthwhile to develop further. Because even though this technique generally will never be quicker than accessing parsed data in memory, it performs well enough, especially if use of caching can be improved. Big steps have been made in memory efficiency, which is promising for the future.

### 5.1.1. MEMORY EFFICIENCY

One big focus point of this project was to successfully parse streaming JSON data without having to load the whole dataset into the memory. For example, some datasets retrieved from various AutoCAD drawings can exceed sizes of 500Mb. Parsing this usually results in loading the whole dataset into memory. For small datasets this would not be an enormous problem, but for large files it most definitely is. Our application can significantly decrease the amount of memory used to access such large JSON datasets. To test this we have used a large JSON file, containing information about all parking lots in San Francisco, and tried to retrieve information from it. This resulted in a memory drop from 185Mb to about 0.4Mb, which is a huge difference. When trying to retrieve more information, thus creating a bigger cache, the result still stays beneath 10Mb.

### 5.1.2. PERFORMANCE OF THE SYSTEM

Whilst the main focus lies more on memory efficiency, the time of the process is also a matter not to forget. The fastest way to search through large JSON datasets, would be to load the entire file in memory and then conduct a search operation. This is of course not the desired solution. We will not escape the fact that speed has to be sacrificed for better memory management. Nevertheless, effort was put in to improve permormance. After refactoring, introducing the concept of lazy loading and implementing a caching technique, the speed of the process changed drastically. This can be seen in figure 3.2. The speed of performing various operations to save memory usage will not easily match the speed of searching through an entire file loaded into the memory.

Having said this, performance testing is an area where we feel we could have done better. We had the ambition to keep track of concrete numbers to measure our performance. We certainly did this on multiple occasions such as when we merged the tokenizer and parser phase. But in retrospect we should have done this throughout the entire project.

## 5.2. FUTURE WORK

Since this project concludes as a proof of concept, there is room for improvement. A solid base has been created, but there are still aspects of the system which can be revamped or extended. In this section we will discuss our view on near-future improvements.

### 5.2.1. MORE CLEVER CACHING

Caching is a very powerful technique and can improve memory performance with large steps. Currently there is only caching on the highest level, i.e. keys and raw values retrieved from a collection are cached. The technique being used is called First Out, First Out (FIFO) which is explained in section 3.3.2. Lots of other techniques exist which are applicable in the current system.

#### TIME BASED REMOVAL

One way to keep the cache clear is to introduce an eviction based on the amount of time an entry is alive. How long an entry should stay in the cache depends on the application for the system. Whether data has to be kept in the memory for a few minutes or for a few hours differs maybe between datasize. Although, a short expiration time could result in more reads from the dataset, which misses the point of a cache. A plus for this technique, is that data is always up-to-date. When the data source is updated, every entry in the cache will eventually be updated as well.

#### FIRST IN, LAST OUT

First in, last out is exact opposite of the FIFO method. This method could be useful when the entries pushed in first are also the most important references. This will, contrary to FIFO, save time looking through the cache for more popular searches.

#### LEAST ACCESSED REMOVAL

Whilst the two caching methods mentioned above are not really usable in this situation, this method is applicable in caching property names with values in the systems context. Least accessed removal means that the cache values that have been accessed the least number of times are evicted first. This technique is similar to the one used in the system right now, FIFO, only now the first one to be removed is the one least accessed by the parser.

A problem to keep in mind using least accessed removal, is that old values in the cache automatically have a higher number of accesses, even if they are not accessed anymore. For example, if the data source has been updated once or twice, the data and starting index could be out-dated. This problem would also occur with FIFO.

### 5.2.2. SUPPORT DIFFERENT BACK END

As this is developed as a library it is important to be as extensible and modular as possible. While we believe that our Parser has good performance and is decently robust, there are other solutions out there as mentioned before. It is worthwhile to investigate integrating these to examine their performance.

### 5.2.3. ERROR REPORTING

We started developing by assuming JSON data was valid, this allowed us to focus on the process. For this library to function in a business environment it is important to properly handle invalid data and provide descriptive error messages, but errors should be easily traceable to the malformed JSON fragment.

### 5.2.4. SUPPORT DIFFERENT DATA FORMATS

There are other data formats that are similar to JSON's key-value pairs. It can be worthwhile to further decouple the functionality from the data format to allow use of structures as YAML or BSON in the future.

### 5.2.5. INTEGRATION IN BUSINESS ENVIRONMENT

This project was developed as a standalone library without any external dependencies. At *Moxio B.V.* a framework is used that warps the current data returned by PHP's native parser. An adapter needs to be created to allow interaction between our parser and *Moxio*'s objects.

### 5.2.6. OPEN SOURCE

Throughout the project we have discussed the possibilities to make this project available to other interested users. By doing this we could reach out to the community who could be curious about our project. Also, if this proof of concept would be finalized, lots of people could use the system for their own use.

# 6

# CONCLUSION

In this chapter we will evaluate the process of our project and draw final conclusions in respectively sections 6.1 and 6.2. In the first section we will reflect on the different stages of our project and evaluate this. After that we will give our opinion on our proof of concept and elaborate on different conclusions.

## 6.1. PROCESS EVALUATION

Throughout this project we have encountered various aspects of the process which worked well, but also some which did not work that well. In this section a short description will be given of the different parts of the process and discuss if they were positive or negative. During the project we used an agile methodology for managing the product development. Our group consists of two members, which makes it very easy to maintain good and constant feedback. At the start of each week, we had a meeting to discuss the weekly planning and deadlines. Next to this we also had a short meeting each morning to reflect on the planning and decide what should be done. At the end of the week we had a retrospect to discuss what we had accomplished that week.

The start of the project was very efficient. From the beginning it was clear that the research phase would be very extensive for our project, because the subject is considerably theoretic. The research phase itself was very informative. Lots of resources were found and known examples were thoroughly examined. A wide spread of information also reflected on a course we had about automata theory. It was good to see that we could use our knowledge directly from one of our courses.

After a good start we were stuck without a starting direction for designing the system. A lot of research was conducted, but the next step was hard to make. We waited too long, before asking for guidance on this dilemma. This obstacle lost us approximately a week in our initial planning. However, this was solved by pushing our deadline a week further because we did not want this small misstep to affect the final product.

The last stage of the project was very efficient. After the next step was clear, and the design was drafted, the implementation went smooth and quickly gained volume. If certain pieces of the implementation did not work out, our agile way of working allowed us to quickly adapt to the situation and design a solution.

### 6.1.1. RESEARCH

Our research phase was relatively long. This was the result of the nature of the project which involved a lot of exploration into automata theory, parser design and implementation and other related projects. The project we were tackling did not have a lot of well defined goals or requirements as you would have in more 'traditional' application development. We think we conducted adequate research to have reached an optimal solution in the end.

We did find it difficult to pinpoint the border between system design and the research phase. Therefore there is for example some overlap between our chapter Design and our research report. Throughout the design

and implementation phases we stumbled upon other concepts or problems with our current approach. This often resulted in some added research.

### 6.1.2. DESIGN DRAFTING

Designing a pull-based library is a bit different than designing a more 'traditional' application. We had no experience with designing such a system, which was a nice challenge. After a good and informative research report, we did not really know what the next step was. Traditionally, it would be easier to grasp such a step by designing an interface for the application and 'filling in' the functionalities. In our case this was not necessary and we struggled with the next step. After searching for counseling with our supervisors, we got that little push in the right direction which we needed. After this everything went a lot smoother.

### 6.1.3. IMPLEMENTATION

The implementation of our design went fairly smooth. Having a clear image of the final product, made the coding process a lot easier. We planned to implement new features in the implementation phase iteratively. Each new feature would improve the system in either performance or speed, so implementing a few features in parallel could cause chaos. This worked very well for the process and made it possible to implement a caching component fully. After each iteration we made sure to have a working product, i.e. completing all tests.

Refactoring caused some unplanned changes in the design, which is not a negative situation. While implementing we came across some performance bottlenecks in the design, which could be solved by refactoring components and thus a changed design.

## 6.2. PROJECT CONCLUSION

At a time where an increasing amount of data is being analyzed, it was very interesting to create a solution which could cope with large data sizes. Especially doing so,= while working within *Moxio*, where web applications are built to facilitate data and information flow. In the past few months we have built a proof of concept, which did not yet exist for these specific circumstances. Doing so, we have implemented a system which can successfully parse large data sets whilst keeping memory usage to a minimum.

At this stage the system is, as mentioned before, a working proof of concept. It is not yet ready to work in a production environment, as some components need expanding. However, we can conclude that the goal set at the start of this project, has been accomplished. The parser can parse streaming data and retrieve information without having to load the whole data source into the memory and using a random access API.

Reflecting on the progress made in almost 3 months, and the results obtained by the parser, it seems worthwhile to continue. Although the system currently is not ready for use in production, it demonstrates the theoretical possibilities for parsing streaming data. The current solution would not need very much more expanding and improving before *Moxio* could use it for their JSON data.

To conclude this report, we can state the project was a success. Still, further improvement have to be made before the proof of concept can be finalized and used in real-world applications.

# A

# INFOSHEET

| | |
|---|---|
| Project Title | Streaming JSON Parser |
| Client organization | Moxio |
| Final presentation date | July 3 2015 |
| Final Report: | TU Delft Respository |

## DESCRIPTION

The amount of data validated and retrieved from technical drawings is ever increasing. In-memory parsing is quickly becoming less viable. Therefore *Moxio* wants to be prepared and wants to find a way to allow processing these amounts of data while maintaining the way this data is structured. Moxio is a business concerning itself with building web applications to facilitate data and information flow from different parties involved with a large building project, e.g. *Spoorzone Delft*, *Rotterdam Centraal*, etc.

To tackle this problem a lot of research has been done in parsing concepts, automata theory and existing parser implementations. This helped us to gain the theoretical knowledge to design involving a pushdown automata to parse raw JSON and an index overlay to efficiently index the data.

A lot of effort was put into maintaining a smooth process. This includes a development environment with tools for automated testing, version control and style checking. But also weekly and short daily meetings to make sure the team was on the same page.

Eventually we delivered a working proof of concept which has a significantly reduced memory footprint. Performance, i.e. speed of parsing, still has room for improvement and would be one of the first things to look at for optimization. It is the intention of *Moxio* B.V. to continue developing this parser and eventually apply it in their own business environment.

## PROJECT TEAM MEMBERS

**T. Booij**
*Interests*: Web development, Computer Science, Security
*Project Role*: Index overlay, caching, parser development

**R.A. Klavers**
*Interests*: Computer Science, Algorithms, Back-end development
*Project Role*: Parser development, refactoring, code clarity

## CLIENT

**M. Wijngaard**
Affiliation Moxio

## COACH

**Dr. ir. A.J.H. Hidders**
Affiliation TU Delft, EEMCS
Web Information Systems

## CONTACT

**T. Booij** timmbooij@gmail.com
**R.A. Klavers** robertklavers@gmail.com

# B

# PLAN OF APPROACH

## B.1. INTRODUCTION

In this working document we will provide a broad overview of our Bachelor Project: 'Streaming JSON Parsing' commissioned by Moxio B. V. . This document will provide an overview of our plan of approach. The first section will introduce the assignment and the client. In the second section we will give an outline of our approach to the problem. The third section we will define the project structure and in the fourth and final section we will discuss ways to ensure code quality.

## B.2. ASSIGNMENT

It is important to get a good grasp on the assignment at hand. In this section we will detail the context of the project by introducing the client and by illustrating the problem.

### B.2.1. CLIENT

Our client, *Moxio*, realizes software products and services where they integrate the domain knowledge of the client. There they have a strong focus on the infrastructure market, with projects like *Spoorzone Delft* or *Rotterdam Centraal*. They work with a passion for technology and knowledge in user-friendly software and realize this in big projects for well-known organizations. Their solutions improve the collaboration and the quality of information in knowledge-intensive products.

### B.2.2. CONTACT INFORMATION

#### TEAM

T. M. Booij *(timmbooij@gmail.com)*
R. A. Klavers *(robertklavers@gmail.com)*

#### PROJECT SUPERVISORS

*TU Delft*
Dr. ir. A. J. H. Hidders *(A.J.H.Hidders@tudelft.nl)*
*Moxio*
M. Wijngaard *(merijn.wijngaard@moxio.com)*

### B.2.3. PROJECT DESCRIPTION

The goal of this assignment is to develop a memory-efficient technique to process and parse large hierarchical JSON data-sets, without conceding in programmatical accessibility of this data, (In other words: Random-access should still be possible) and implement this within the environment of applications used by *Moxio*.

Traditionally these JSON data sets are fully parsed and loaded in memory. More often than not just a part of this data is directly used in the application. As these data-sets are only growing in size, the need arises to look for methods to process this data more efficiently. *Moxio* is interested in ways to keep using the current practises, i.e. "pretending" the data is in memory by using traditional random access, while working behind the scenes to fetch data on such a way to keep as little in memory as possible.

## B.3. APPROACH

In this section we will discuss our approach to the project. We will go over our planning, development methods and requirements drafting.

### B.3.1. REQUIREMENTS

We use the research phase to get a better idea of the subject at hand and draft the requirements to make this project a success. The initial requirements are not yet explicitly defined so it is important to get a good idea of the direction we need to go. We will make use of the MoSCoW model to define what requirements are needed to bring our project to fruition.

### B.3.2. AGILE WORKFLOW

As the requirements are not all set in stone, it's important to keep a flexible approach during development and being able to change direction if a certain approach proves unfeasible. Therefore we choose to adopt an agile workflow, meaning short sprints of 2/3 weeks with frequent evaluation to be able to adapt to changing circumstances. As we're just a team of two we found it unnecessary to adopt a full Scrum paradigm, we will make sure to meet regularly with the client to ensure we're still on the right track.

### B.3.3. PLANNING

We have set up a global planning which splits the project in different phases. To make sure we reach the deadlines set for the deliverables we drafted a strict planning. We also made a broader overview to serve as a guideline to what we should be doing in a given week.

#### DELIVERABLES

| | | |
|---|---|---|
| **Week 1** | *30/04* | Plan of Approach |
| **Week 2** | *07/05* | Research report |
| **Week 7** | *10/06* | SIG evaluation 1 |
| **Week 9** | *24/06* | SIG evaluation 2 |
| **Week 11** | *09/07* | Final presentation |

#### RESEARCH PHASE

Week 1: Creating a Plan of Approach and begin researching subject matter.
Week 2: Finishing research report.

#### IMPLEMENTATION PHASE

Week 3-4: First sprint
Week 5-6: Second sprint
Week 7-8: Third sprint, First SIG review

#### REPORT AND PRESENTATION

Week 9: Last implementation finishing touches, Process SIG review, first draft of Final Report
Week 10: Final SIG review, finish Final Report
Week 11: Final Presentation

## B.4. PROJECT STRUCTURE

In this section we will go over the project organization. The team will be introduced and the development process at hand will be discussed.

### B.4.1. MEMBERS

The members of the project are Robert Klavers and Tim Booij. Both will work at least 40 hours per week to ensure the mandatory 15 EC are satisfied. The labor is evenly dispersed across all deliverables (documentation, implementation, etc.).

### B.4.2. REPORTING

In this subsection we will discuss the administrative processes we will use to make sure the communication will be smooth.

#### MONDAY MEETINGS

Each Monday we will have a meeting with each other to determine this weeks planning. For each week we will assess what we will achieve in that week and create an agenda. To keep track of any tasks, we will use an application called *Wunderlist* for our project management. This will also be cleaned up at the start of each week.

#### DEVELOPMENT PERIOD

During the development phase of our project, we will have daily meetings to discuss what problems we will tackle that specific day. As we are a group consisting of only two members, the communication is fairly easy.

#### REPORTING TO THE CLIENT AND SUPERVISOR

Each week we will report our progress to the client in a meeting and discuss this. We will send our supervisor updates at least each two weeks. The deliverables will also be sent to the supervisor for review when needed.

## B.5. QUALITY ASSURANCE

As in any software development process it's important to follow a set of rules to make sure code quality is guaranteed. Although we do not yet know what technologies we will be using, the principles will remain the same. We will make sure the code is adequately tested by writing unit tests. We will review each others code to ensure there's always more than one pair of eyes sees a particular piece of code. We will also make use of Git as our version control.

# C

# Research Report

This report is the result of a two-week phase of intensive research in the problem domain. This chapter will outline this phase by going over each aspect of our research. The goal of the research phase was to get a good understanding of the problem and to be able to define it. In section C.1 we will introduce the problem and properly specify and analyze it. Before we can define the goals and requirements of our solution we first need a good understanding of the techniques involved. Therefore we also introduce a few technologies involved. With this knowledge we can, in cooperation with the client determine requirements for the project in section C.2.

It's important to know what else has already been researched or been developed before we start implementing our own solution. In section C.4 we provide an overview of similar problems and their solution, but also why they are not directly applicable in our situation. In the next section, section C.5 we introduce the direction we believe we need to go in. Following on that we will discuss how to get there by going over our development process and ways of evaluating the performance of our solution in section C.5.2 and section C.5.3 respectively.

## C.1. Introduction

### C.1.1. Problem definition

In a major part of their applications *Moxio* deals with a lot of data from a variety of sources in the building sector. A large focus point is supporting data-transfers during large projects. For example, *Spoorzone Delft* or *Rotterdam Centraal*. *Moxio* gathers data from various sources to combine them and test them for consistency and conformity according to a specific rule set. These are sources like MicroStation, SAP and ArcGIS. To deal with this diversity in data formats and sources they use multiple applications written in different languages. Communication between these applications is done through HTTP requests sending JSON packages. There is a growing trend where these packages keep getting larger. Therefore it is soon no longer efficient to store this data in memory.

Current solutions for this problem often involve streaming solutions, which quickly becomes impractical when dealing with highly interconnected or hierarchical data. Therefore we're trying to find the best of both worlds: The memory-efficient technique of sequential streaming parsing, while still being able to access this data in a traditional way.

### C.1.2. JSON

JSON, or JavaScript Object Notation, is a lightweight data-interchange format that gives us a human-readable collection of data that we can read and access in a logical manner. Basically it is used to transmit data objects which consist of attribute-value pairs. JSON originates from the need for a way to to transmit data between a web application and the server it has to communicate with. In the early 2000's Flash and Java applets were the obvious way for real-time server-to-browser communication. To make this a lot easier and generic Douglas

Crockford specified the JSON text format to facilitate structured data interchange between all programming languages. JSON was inspired by the object literals of JavaScript, or EMCAScript as defined in the ESMAScript Language Specification, and is a syntax which consist of commas, colons, brackets and braces. See listing 5 for an example [1].

The text format is applicable in many different applications and contexts. Normally programming languages vary in support for objects, but when they do the chances are also very high that the characteristics of these objects are different. Essentially the models of object-based systems can be very divergent and are also constantly evolving. JSON on the other hand, provides a simple standard notation for attribute-value pairs [2].

```
{
    "firstName": "John",
    "lastName" : "Smith",
    "age" : 30
}
```

Listing 5: JSON example

### C.1.3. XML WORLD

Extensible Markup Language, or XML, is considered the 'holy grail' of computing, because of its unique and universal data representation. It is a markup language which defines a set of rules for encoding documents readable by both human and machine.

A simple way to explain XML would be to explain that it is a restricted subset of the Standard Generalized Markup Language (SGML) developed in the 1960s. The SMGL is the standard for document description. It is designed to enable text interchange. Documents following the SMGL markup have a precisely described structure that may be interpreted by computers and understood by humans. These documents should not specify the processing to be executed on it, because a declarative structure is less likely to have [3].

The goal for XML was to create a markup language which could be received and processed on the web by HTML. XML was designed to be easy to implement and operate with SGML and HTML. The primary applications of XML are Remote Procedure Calls and object serialization for data transfer between applications. An example of an XML structure can be seen in listing 6. In this example it is clear that XML does not use predefined tags. Each set of tags is created by either a human or a computer [4, 5].

```
<email>
    <to>John</to>
    <from>Sam</from>
    <heading>Note</heading>
    <body>Don't forget our meeting!</body>
</email>
```

Listing 6: XML example

### C.1.4. A COMPARISON

In a way, JSON and XML are both just another approach to represent data. XML being simpler then the SGML and JSON respectfully simpler then XML. Why just not use XML, as it is easy to understand for both humans and computers? There are several reasons for this. A case study from 2009 shows the differences between the two data interchange formats. This case study uses test cases which are designed and implemented to compare transmission times and resource utilizations of XML and JSON. In these test cases a client sends different amounts of objects to a server and compares the time and resources used for it. The results show that JSON is faster and uses less resources then XML. Both provide their unique power, but the gravity of resource utilization and performance must be assessed to make a decision for a data interchange format [4].

## C.2. REQUIREMENTS

In this section we will describe the goals we will set for our parser and the requirements needed to achieve this. First we will give an explanation of the data which this problem has to deals with. Then we will explain what prerequisites are needed to build a parser that can deal with the given data in the way the problem describes.

### C.2.1. DATA EXAMPLE

To get a better feeling for the kind of data we're dealing with, we have constructed an example. It is important to note that starting out our solution shouldn't be data driven, i.e. the structure of the data shouldn't influence the implementation.

At *Moxio* a lot of effort is put into validating data extracted from AutoCAD drawings. In figure C.1 an example is given of one of such AutoCAD drawings: A cross section of the platform of *Spoorzone Delft*. An AutoCAD drawing consists of a number of Blocks, each block has a specific handle, i.e. id, and a list of entities which also have a handle and refer another Block. listing 7 gives an example of this data. Due to numerous references to different blocks it is clear that this data can by highly interconnected.

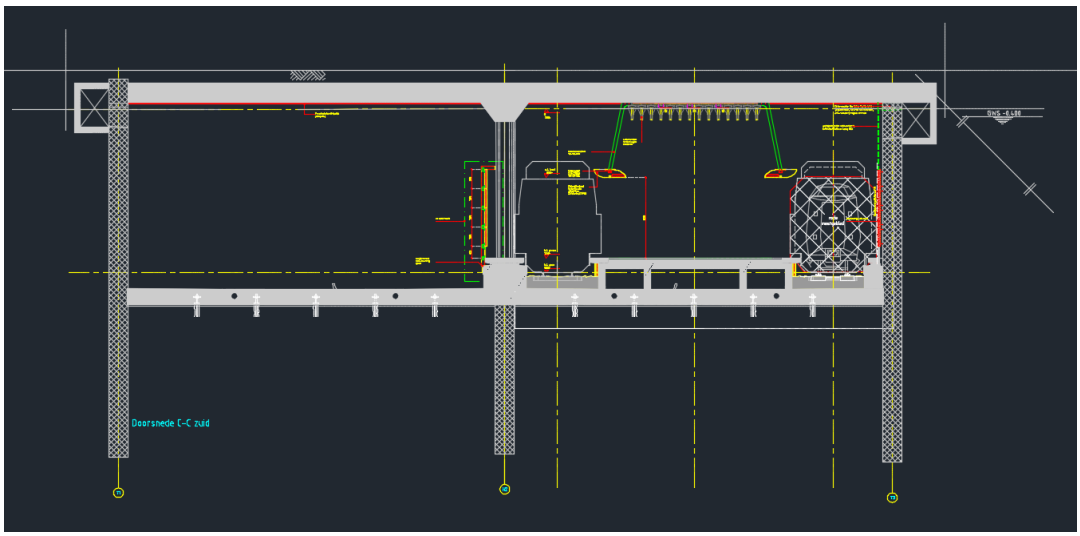


Figure C.1: Example drawing Spoorzone Delft

Immediately it becomes clear that there can be a lot of interconnectivity, there are lots of references in-between "Blocks" which means that just one pass through the data might not always be enough.

### C.2.2. DESIGN GOALS

#### STARTING OFF

While implementing our solution, we always want to have a working version. That is why we have decided to work with different milestones, with each milestone being a further improvement to our solution. We want to start with a really simple implementation of our parser. This implementation would not be efficient or extremely fast. We will explain this, and the milestones, later on in section C.5.2. After this beginning we will add features and improve it step by step.

#### RANDOM ACCESS

An important requirement of our implementation is that it should use random access. Basically there are lots of ways to categorize parsers. When creating a high-performance parser you can choose to build a sequential access parser or a random access parser. When data is sequentially accessed, the data processor only had to access the data currently being looked at. The parser does not have access to data earlier specified and cannot

```
{
    "Blocks": [
        {
            "Entities": [
                {
                    "Handle": "242E6",
                    "Type": [
                        "Block Reference",
                        {
                            "Attributes": [],
                            "Block Handle": "23E01"
                        }
                    ]
                },
                {
                    "Handle": "242E7",
                    "Type": [
                        "Block Reference",
                        {
                            "Attributes": [],
                            "Block Handle": "23E01"
                        }
                    ]
                },
                ...
            ],
            "Handle": "C454",
            "Name": "xref-station-zuid"
        },
        {
            ...
        }
    ]
}
```

Listing 7: Example data retrieved from AutoCAD drawing

look ahead either. These types of parsers are known as event based parsers. The SAX parser is a well-known example, which we will discuss in section C.4. Sequential access limits parsers to the 'window' that was just parsed. This is not what is needed for the data at hand, it always has to be possible to access data which has been parsed earlier or ahead. This is where random access parsing has the advantage. As the name says it; random access parsers give the possibility to access data at random, so a processor can move back and forth over the data being parsed [6]. See figure C.2 for a visual representation of both parsers.

A drawback however, is that random access parser implementations often are slower than sequential access parsers. The general idea of a random access parser, is to build an object tree consisting of the already parsed data. This makes it possibles to access other parsed data, but also needs some computational power and can reserve some memory. We want to use as little memory as possible, so creating a tree would not be efficient enough. Another solution would be to create indices to point the start and end of elements found in the parsed data [6]. With these indices, the parsed data can be accessed without having to save the objects in a tree.

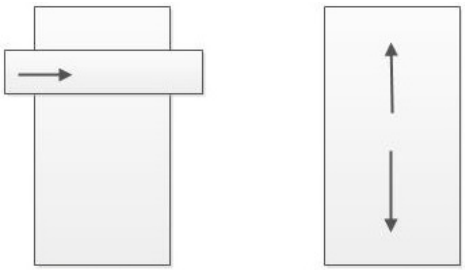

Figure C.2: Sequential access and random access

## C.3. PARSING

We found it important to first get a better theoretical understanding before moving on to specific implementations to be able to motivate design decisions and better grasp existing solutions and their implementation. This allows us to put various techniques and ideas better into perspective, even though perhaps not directly related to our problem. In this section we will briefly outline the concept of parsing and various techniques that have been developed.

### C.3.1. PARSING CONCEPTS

Parsing is the process of splitting up raw data into its separate components. Parsing is an important step in a compiler, taking tokens obtained from lexical analysis and passing it on to later stages of the compiler. However parsing is also important for other processes and languages. For example when you point your web browser to a HTML page or when you open a .docx document. Data structures like XML and JSON also need to go through a parser before you can access this data in your application.

In general every parser tries to build a parse tree, i.e. it will try to generate the input from the grammar.[7] There are two major ways to achieve this, top-down parsing and bottom-up parsing. Top-down parsers create a parse tree from the root, while trying to infer the necessary production based on the next input token. A bottom-up parser will start from the leaves and check if a set of leaves can be combined. Both these methods can work in linear time on context-free grammars. [8]

As mentioned in section C.2 the implementation can differ. For instance, by default PHP parses a JSON file by using a pushdown automaton, i.e. it is a finite state machine using a stack to keep track of state. [9] It checks the validity of the data and creates a PHP variable along the way. This means that it will load the JSON file into memory and expose it to the programmer an stdClass or Associative Array. Most programming languages have a similar parser available that will parse a JSON file into an equivalent native data structure.

### C.3.2. STREAMING PARSING

A problem arises when the amount of JSON data becomes too large to be able to conveniently handle it through this native API. Instead of loading everything in memory you need a way to just be able to access the parts you need while discarding the rest. A way to solve this is to stream your data and only act if you encounter something you need. Two distinct ways to do this are pull- and push based parsing.

Pull parsing follows the iterator design pattern. You control the flow of the parsing and can invoke methods when needed. With push parsing you receive events from the parser and can act upon those using callbacks for each respective JSON element type. Both these techniques require a different approach than recursive descent parsing. To determine the next element type it is necessary to keep state to deduce its context. In other words, a pushdown automaton is able to properly parse the data and check for syntactical correctness. A nondeterministic pushdown automaton is able to recognize all context-free languages [10]. As JSON is a context-free language, i.e. for the JSON grammar the left-hand side of its production rules consists only of a single nonterminal symbol.

### JSON PARSING CONTINUED

These production rules are defined by ECMA-404 "The JSON Data Interchange Standard" [11]. There are other specifications, such as The JavaScript Object Notation (JSON) Data Interchange Format, drafted by the Internet Engineering Task Force (IETF) in March 2014[12]. These are syntactically identical, the IETF specification cleans up a number of ambiguities and inconsistencies and provides a number of recommended practises, in order to prevent situations allowed by the specification but can cause problems in the real world.

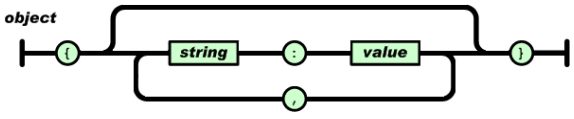


Figure C.3: JSON Object

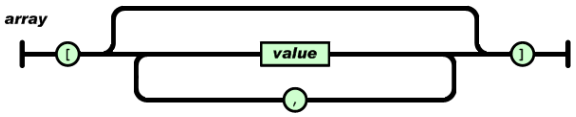


Figure C.4: JSON Array

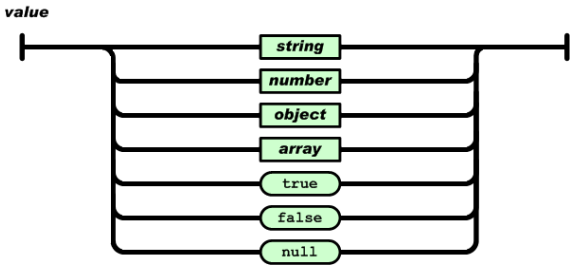


Figure C.5: JSON Value

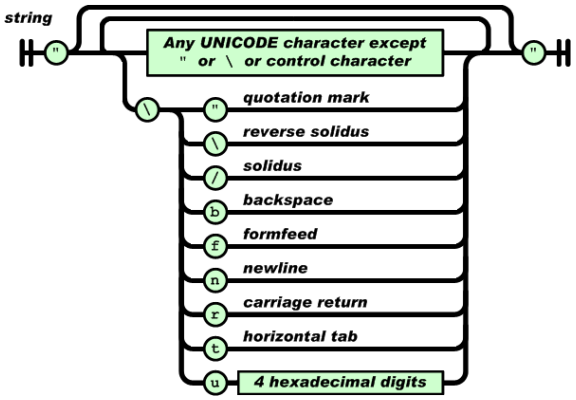


Figure C.6: JSON String

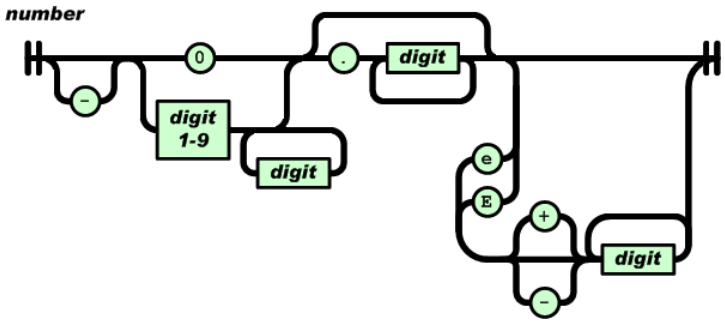


Figure C.7: JSON Number

**Pushdown Example** As mentioned a pushdown automaton is able to recognize context-free languages, i.e. accept a word if it is a part of the language or discard if it is not. A language consists of all words that can be generated from a formal grammar. To illustrate this take the following grammar:

$$G = (\{S\}, \{(,)\}, P, S)$$

where $P$ is the set of production rules:

$$\begin{aligned} [!ht]\, S &\rightarrow SS \\ S &\rightarrow (S) \\ S &\rightarrow () \end{aligned}$$

This example can be used to match opening brackets to closing brackets. This can be expanded to include arrays, variables, strings, etc.

**Property Names** There is however one problem with considering JSON as a context-free language. There is no way to guarantee that property names are unique. Identical property names are allowed by ECMA-404, obviously it is still syntactically valid. Whereas RFC7159 states "The names within an object SHOULD be unique."[12]. In practise it's up to the library to determine what happens, most libraries deserialize into a hash map or dictionary where uniqueness of keys is guaranteed. Others take the last occurrence of the name and use that instance.

## C.4. SIMILAR TECHNOLOGIES

### C.4.1. EXISTING PARSERS

The concept of a parser as outlined in section C.3.2 is not new. There are existing parsers which follow a similar pattern. There are various reasons why these were not suitable for our case. In general we set out to keep external dependencies as low as possible, this due to the business environment and the extra overhead it would cause. Also not insignificant is that we really liked the challenge of writing our own parser. But most of all, the solutions we did find proved inadequate.

As touched upon in section C.3 there are three different parsing models: "traditional" in-memory random access, pull-based parsing and push parsing. The XML world is a mature field and lends itself well to further illustrate this

#### SAX AND STAX

SAX (Simple API for XML) is a well-known push-based (event-driven) parser for XML documents. Being a push parser you need to define callbacks to act on the incoming data. By contract, a major pull-based parser is StAX (Streaming API for XML) where you, as a client, can request XML data when you explicitly ask for it. An advantage of StAX (and pull-based in general) is that you do not need to keep state to process more complex structures. Instead the state is simply kept on the stack and allows for a more natural approach.

Naturally the JSON environment is less developed, nevertheless these two approaches have their JSON counterparts. For example Jackson is a multi-purpose Java library for processing JSON, which is inspired by XML tooling. [13]

#### EXT-JSONREADER

`ext-jsonreader` is an extension for PHP developed by Shahar Evron. Initially this project looked really promising and quite close to what we needed as a back-end for our system. Nevertheless there were a number of reasons why we decided to create our own solution. First of all we'd like to keep as much control on the parsing phase as possible. Because this is an extension based on a C library the functionality is hard to extend or change. Secondly, as mentioned in their readme the extension is experimental, had no internal error handling and hasn't properly been updated in 6 years.

#### SALSIFY JSONSTREAMINGPARSER

This parser uses a push-based design, analogous to XML SAX. We already discussed why this is not suitable for our problem, but we found it deserved a mention. Besides `ext-jsonreader` this was the only other mature implementation of a streaming JSON parsing solution we found for PHP.

### C.4.2. INDEX OVERLAY

While being the go-to solution for low-memory parsing, the streaming model is less suited for our problem. As mentioned the problem we face requires moving back and forth through the data. Therefore it's unfeasible

to use a streaming approach, these methods shine when one pass is enough to process your data, e.g. transformation, validation. Since a major precondition of the project is to keep using the current API, i.e. using PHP's ArrayAccess interface, we started looking at other options.

Another way to reduce memory load is to refrain from storing all data in your parse tree but to only keep a reference to the location of the element. Also known as an Index Overlay. A similar idea has been a big success for VTD-XML (Virtual Token Descriptor for eXtensible Markup Language). VTD-XML is a collection of XML processing technologies which uses something similar to an Index Overlay to access the XML data. VTD uses a 64-bit integer to encode the offset, length, token type and nesting depth of an XML element in an XML document. [14]

This has the added benefit of being able to construct the index in a lazy way. You only need to construct parts of the tree actually used at that time. While creating a decrease in performance, this will greatly cut down on memory usage.

## C.5. APPROACHING THE PROBLEM

Having explored the theory and existing solutions for efficient (JSON) parsing we can start devising our own approach. This section will go over the solution we found most feasible at the start of development and will motivate this.

### C.5.1. CONTRADICTION ISSUE

At a glance it seems counter intuitive to both maintain a small memory footprint while maintaining random access to your data. It's difficult to come up with a solution which performs more efficient and is able to keep functioning in the current environment at *Moxio*. As argued above a Document-Centric approach covered in section C.4.2 combined with a lazy approach to data retrieval will maintain this random access while having a greatly reduced memory footprint compared to traditional parsing, at the cost of a potential loss in performance. Since the primary concern is reduced memory usage and not increase in performance, we set out to take this as a starting point.

Starting out you can identify two major components: Parsing and the Index Overlay structure.

We identified a number phases or processes in the system we envision: Parsing the data, creating an index overlay atop of the parsed data and a mechanism to cache this data to allow for quick retrieval.

#### PARSING

The parsing component is responsible for translating raw text data into semantic data on the JSON elements present. In section C.3 we mentioned that tokens retrieved from lexical analysis get passed on to the parser. The same will need to happen here, raw data gets 'tokenized', these tokens will be parsed to give them meaning, e.g. *Object start, property name, string value, etc.* The actual data, e.i. keys and values, are actually not that interesting, we merely require a stream of parsed JSON Elements.

Traditionally these parsed elements will then be used to construct the parse tree, however in our case this will be done on a call-by-need basis. Similar to the pull-based approach as discussed in section C.3.2.

A major difference compared to this approach is the need to be able to seek through the data, since we just want store the index and length of an element the parser needs to be able to start from any given position in the data. Or rephrased, the parser has to be able to work on a scoped subset.

In summary, the parser will act on a seekable datastream and will be able iterate over JSON elements from a given index in the data.

#### INDEX OVERLAY STRUCTURE

As we discovered in section C.4.2, having an index overlay provides a nice trade-off between not loading anything and loading everything in memory. This overlay will store the start index and length for a given key,

i.e. it will have to iterate over the elements parsed by the parser and create an overlayobject to allow data retrieval.

### C.5.2. DEVELOPMENT APPROACH

Due to exploratory nature of the problem assignment we quickly decided that we had to take an agile approach to development. As we're a two man team a full scrum approach felt unnecessary and provides too much overhead. Nevertheless it is important to have a clear image of what to work on and keep the big picture in mind.

### C.5.3. PERFORMANCE TESTING

Closely related to keeping an agile approach to development is having the ability to be able validate and evaluate implemented solutions. So it's important to create a system at the start that allows us to actually compare our solution to its previous iterations and to already existing solutions in both memory usage and general performance.

## C.6. CONCLUSION

During our research phase we were able to further define the problem, and with this in mind set a number of concrete requirements. Having gained theoretical knowledge of parsing concepts and having acquired an overview of existing systems and techniques gave us direction. With this knowledge we can develop a more concrete system design which we will specify in our final report.

# D

# Software Improvement Group Feedback

In this working document we will elaborate on the feedback received from the Software Improvement Group. We will give our own comments on the feedback from SIG and reflect on their suggestions.

## D.1. First Submission

We submitted our codebase for the first time 8 weeks into the project. At this point, we had a working parser. Not everything was as neatly implemented, but nevertheless it had a solid base.

### D.1.1. Feedback

*[Analyse]*

*De code van het systeem scoort 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door lagere deelscores voor Unit Size en Unit Complexity.*

*Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt.*

*Een aantal van jullie langere methodes schalen niet zo goed op het moment dat je project verder gaat groeien.* `TokenIterator.getToken()` *is hier een goed voorbeeld van, zo'n grote switch is moeilijk te lezen en moeilijk te testen. Daarbij lijken de cases sterk op elkaar, waardoor je kans op copy/paste fouten krijgt. Het enige wat jullie in deze methode doen is een nieuwe Token maken op basis van de input. In plaats van deze switch zou je dat ook anders kunnen doen, bijvoorbeeld met een map van de letter naar het bijbehorende token type.*

*Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Ook hier geldt dat het opsplitsen van dit soort methodes in kleinere stukken ervoor zorgt dat elk onderdeel makkelijker te begrijpen, makkelijker te testen en daardoor eenvoudiger te onderhouden wordt. In dit geval komen de meest complexe methoden ook naar voren als de langste methoden, zoals bijvoorbeeld* `JsonParser.next()`*, waardoor het oplossen van het eerste probleem ook dit probleem zal verhelpen.*

*Complimenten voor de hoeveelheid testcode die door jullie is geschreven. De verhouding tussen de hoeveelheid productiecode en de hoeveelheid tests ziet er erg gezond uit, en deze manier van test-driven werken gaat jullie op termijn voordelen brengen op het gebied van onderhoudbaarheid en stabiliteit.*

*Ondanks bovenstaande punten scoort de code dus bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.*

### D.1.2. COMMENTS

It is nice to see that our efforts to produce well tested functions get noticed as such. It was especially good to see that the feedback also reflected our progress up until then. We proceeded to implement the design without this feedback, and after further improvements the feedback reflected what we had done until that point. We agree that methods like `TokenIterator.getToken()` and `JsonParser.next()` were quite large and should be our primary focus for refactoring. Summed up we are pleased so far, especially with the stage of development at this time.

## D.2. SECOND SUBMISSION

Our second submission was made 10 weeks in, our codebase at this time was in a near finished stage, with a lot of effort put into refactoring, test completeness, i.e. finding edge cases and making sure they're properly covered, and general code clarity and readability.

### D.2.1. FEEDBACK

*[Hermeting]*

*In de nieuwe upload zien we dat het codevolume bijna is verdubbeld, terwijl de score voor onderhoudbaarheid ongeveer gelijk is gebleven.*

*Op het gebied van Unit Size en Unit Complexity, die bij de vorige analyse als verbeterpunt werden aangemerkt, zien we nu een lichte verbetering. Jullie hebben een aantal van de genoemde voorbeelden aangepakt, maar deze aanpassingen zijn niet structureel genoeg om een invloed op de totaalscore te hebben.*

*Een andere reden dat jullie score niet verder is gestegen is de duplicatie. Sinds de vorige upload zijn er een aantal blokken code gedupliceerd, bijvoorbeeld in* `IndexOverlayArrayElement.php`*. Probeer in de toekomst dit op een meer generieke manier op te lossen.*

*Het is positief dat jullie naast nieuwe productiecode ook nieuwe testcode hebben geschreven. Ondanks de tijdsdruk hebben jullie dus vast weten te houden aan jullie eerdere manier van werken, complimenten.*

*Uit bovenstaande observaties kunnen we concluderen dat de opmerkingen uit de eerdere evaluatie zijn meegenomen in het ontwikkeltraject.*

### D.2.2. COMMENTS

Again we're quite pleased with the result, we agree on the two main 'problem' areas, being Unit Size and Complexity and code duplication. The examples given by SIG were two areas we also marked for improvement. We tried to keep adequate tests throughout the whole process, so it's good to see that get noticed.

# BIBLIOGRAPHY


[1] D. Crockford, *The application/json media type for javascript object notation (json),* (2006).

[2] ECMA International, *Standard ECMA-262 - ECMAScript Language Specification*, 5th ed. (2011).

[3] E. Van Herwijnen, *Practical sgml* (Springer Science & Business Media, 1994).

[4] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, *Comparison of json and xml data interchange formats: A case study.* Caine **2009**, 157 (2009).

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, *Extensible markup language (xml),* World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210 , 16 (1998).

[6] J. Jenkov, *Implementing high performance parsers in java,* (2013).

[7] M. Lam, R. Sethi, J. Ullman, and A. Aho, *Compilers: Principles, Techniques and Tools*, 2nd ed. (2006).

[8] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master* (Addison-Wesley Professional, 2000).

[9] *Php source code,* .

[10] M. Sipser, *Introduction to the theory of Computation*, 3rd ed. (2013).

[11] E. International, *Standard ECMA-404 - The JSON Data Interchange Format*, 1st ed. (2013).

[12] I. E. T. F. (IETF), *The JavaScript Object Notation (JSON) Data Interchange Format* (2014).

[13] *Jackson json processor,* .

[14] J. Zhang, *Vtd: A technical perspective,* .