UNIVERSITY OF OSLO
Department of Informatics

A Javascript API for accessing Semantic Web

Master Thesis

Arne Hassel

Spring 2012

Acknowledgments

[To Be Written (TBW)]

Abstract

[TBW]

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The Semantic Web (SW) is a many-faced entity, a colossal structure of standards and resources. It's also an idea shared by a multitude of communities, a concept of structured information, and an abstraction of knowledge. It's a mixture of technologies, created over a decade of work by professionals. Academia researches it, businesses try to create common ground with it, and visionaries preach of it's promises: A richer world, where computer-driven agents find, process, and act upon information tailored for our need.

At the center of the SW we have the World Wide Web Consortium (W3C), led by Tim Berners-Lee. Berners-Lee's perhaps more famous for his invention, the World Wide Web (WWW), and he's also the one who coined the phrase Semantic Web. It's in his writings of Design Issues we find the essence of SW, namely the sentence "The Semantic Web is a web of data, in some ways like a global database" [**?**].

The web of data has been in the making since the late 90s, but in terms of traction there's still much to be done. Some complain it's still very much an academic affair, while others complain of the lack of interest from the developing community.

This master thesis has taken the approach to look at the gap between SW and the developing community by trying to construct a framework that offers tools to access the SW. It's been written in and for Javascript (JS), as it is a programming language of the web, and the time seems right.

JS can relate to SWs struggles for traction. For long time it was ridiculed by developers, saying it was a silly language that merely created fancy effects on web pages, but not doing anything useful. Douglas Crockford, a JS-evangelist, has called JS the world's most misunderstood language [**?**]. And if the name and it's syntax wasn't confusing enought, the browsers with their differing implementations weren't making it any easier.

There were, and still are, many reasons to why people get confused by JS. But in the mid-2000s, efforts were made to make JS more accessible to developers. Prototype, MooTools, and jQuery are all frameworks that promises APIs for easier, cross-browser access to the power within JS. And it worked! Readily manipulation of the Document Object Model (DOM), asynchronous

fetching of resources with Asynchronous Javascript and XML (AJAX), and the increasing effort of making JS into a full-fledged server-side programming language, are making JS a powerful and fun tool for developers to work with.

It is this fertile ground the work of this master thesis is trying to tap into. graphitejs (Graphite) is a Asynchronous Module Definition (AMD)-based framework (described in section 2.2.6.1) written in JS that sports a modularized Application Programming Interface (API) to fetch resources in the SW, process it and output in useful way for JS-developers. And perhaps it will help spark the fume that the standards of SW have set, so that we may answer the question: Is SW ready for broad adoptation?

[To Be Written (TBW): What are the thesis' contributions?]

This master thesis will describe the work and choices made during the implementation of Graphite. It's divided into two parts. The first consists of the underlying theory and constraints in technology (chapter 2), how this fits into the scope of this thesis (chapter 3), which tools we made use of (chapter 4), and finally an extensive presentation of the framework itself (chapter 5). The second part offers a discussion of the work (chapter 6), and a conclusion of the matter (chapter 7).

Chapter 2

Background

This chapter will describe the technologies, standards, and theories that Graphite has been build upon.

2.1 Semantic Web (SW)

SW is many things to many people, and seeing the whole picture may not always be so easy. A perhaps fitting metaphor is the story of the elephant and the blind men. It's a story made famous by the poet John Godfrey Saxe, and tells the story of how six men tried to describe an elephant. Depending on which part they touched, each described the elephant differently. One approached its side, and called it a wall. Another touched the tusk, and surely it had to be a spear. The third took hold on the tusk, and spoke of how it resembled a snake. The fourth reached out for its knee, and stated it had to be like a tree. The fifth touched the ear, and ment it had to be like a fan. Finally, the last one had grabbed its tail, and stated how it had to be like a rope [**?**]. As such, what are some of the descriptions we have of SW?

- A web of data [**?**]
- An extension of WWW [**?**]
- A `killer` app [**?**]
- W3C's vision of the Web of linked data [**?**]

[TBW: Logic for the masses; Get a reference to the logical features of SW]

The list above are some of the descriptions in literature, and they're all true. Other aspects of SW is the set of standards it sports (e.g. Resource Description Framework (RDF), Resource Description Framework Scheme (RDFS), Web Ontology Language (OWL), and SPARQL Protocol

and RDF Query Language (SPARQL)), technological foundations (e.g. Linked Data (LD)), applicabilities (e.g. use of Linked Open Data (LOD) amongst governments), social consequences (democratizing data), limitations (e.g. Anyone can say Anything about Anything (AAA)), and more.

2.1.1 Resource Description Framework (RDF)

At the heart of SW lies RDF. It's a formalized data model that asserts information with statements that together naturally form a directed graph. Each statement consists of one subject, one predicate, and one object, and are hence also often called a triple. The three elements have meanings that are analogous to their meaning in normal English grammer [**?**, p. 68-69], i.e. the subject in a statement is the thing that statement describes. An example of statements and how they are represented as a graph is showed in figure 2.1.

- Arne knows Kjetil
- Arne has lastname Hassel

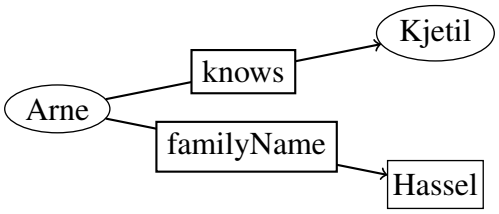

Figure 2.1: A directed graph showing that the subject "Arne" is related to the object "Kjetil" by the predicate "knows", and to the object "Hassel" by the predicate "familyName".

You might've noticed that the two objects have different shapes, one being a circle (like the subject), and the other being a rectangle. That is to show that "Hassel" is a literal. Literals are concrete data values, like numbers and strings, and cannot be the subjects of statements, only the objects [**?**, p. 69].

The circles on the other hand, are known as resources, and can represent anything that can be named. As RDF is optimized for distribution of data on WWW, the resources are represented with Internationalized Resource Identifiifiers (IRIs) (IRI is an extension of Unified Resource Identifier (URI), and is explained in section 2.1.4.2).

IRIs are usually declared into namespaces, to make terms more human-readible (e.g. resources in the namespace http://example.org/ could be prefixed ex). If we look at figure 2.1, we have two resources, namely Arne and Kjetil. To make these available as LD, we could assign them into the namespace `ex`, writing them respectively as ex:Arne and ex:Kjetil.

The basic syntax in RDF has a relatively minimal set of terms. It enables typing, reification, various types of containers (bags, sequences, and alternatives), and assigning of language or datatype to a literal [**?**]. It's power lies in its extensibility by URI-based vocabularies [**?**]. By sharing vocabularies as standards between software applications, you can easier exchange data.

With this in mind, we see that figure 2.1 is faulty, and we turn to figure 2.2 to see a correct representation (using the vocabulary Friend of a Friend (FOAF), prefixed `foaf`, for the properties).

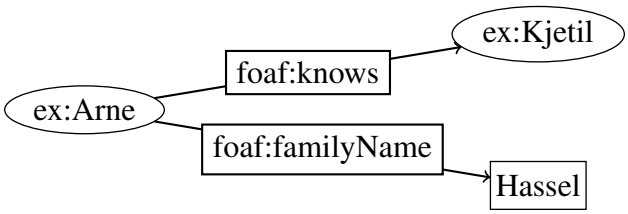

Figure 2.2: Statements from figure 2.1 correctly represented with IRIs.

Not all resources are given IRIs though. The exception to the rule are Blank Nodes (BNs), which represent resources that haven't any separate form of identification [**?**] (either because they cannot be named, or it isn't possible and/or necessary at the time of modelling). These resources aren't designed to link data, but to model relations of resources that are given IRIs.

An example of modelling BN is given in figure 2.3, where we've modelled that ex:Arne has a friend, who we don't know anything about except his nicks, Bjarne and Buddy.

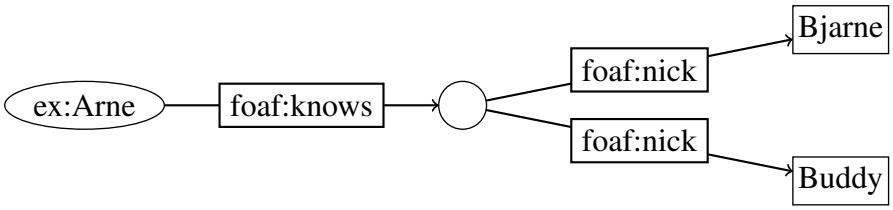

Figure 2.3: A graph containing a BN.

The figures 2.2 and 2.3 are examples of the form of visualization we'll have of RDF-graphs.

2.1.2 Resource Description Framework Scheme (RDFS)

RDFS is an extension in form of vocabulary that extends the semantic expressiveness of RDF. But RDFS is not a vocabulary in the traditional sense that it covers any topic-specific domain [**?**, p. 46]. It's designed to extend the semantic capabilities of RDF, and in that sense it can be regarded as a meta-vocabulary.

The perhaps most important feature of RDFS is its ability to support taxonomies. It empowers the use of rdf:type by introducing rdfs:Class, in effect enabling classification. The properties rdfs:range, rdfs:domain, rdfs:subClassOf, and rdfs:subPropertyOf further extends this feature.

It also builds on the reification-properties of RDF, by instantiating rdf:Statement as a rdfs:Class. It continues by clearifying the semantics of rdf:subject, rdf:predicate, and rdf:object by instanciating them as rdf:Property, and in terms of entailment (explained in section 2.1.7) ties together with rdfs:range and rdfs:domain.

Another extension is the clearification of containers by introducing the class rdfs:Container

and the property rdfs:containerMembershipProperty, which is an rdfs:subPropertyOf of the rdfs:member [**?**].

Finally, it introduces the utility properties rdfs:seeAlso and rdfs:isDefinedBy. The former represents resources that might provide additional information about the subject resource, while the latter gives the resource which defines a given subject. It also clearifies the use of rdf:value, to encourage its use in common idioms [**?**].

2.1.3 Web Ontology Language (OWL)

In the same way RDFS is an extension to RDF in order to express richer semantics, OWL is an extension to RDFS to express even richer semantics. It does so by introducing vocabularies that are based on formal logic, and aims to describe relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer type of properties, characteristics of properties (e.g. symmetry), and enumerated classes [**?**, sec. 1.2].

As of this writing, OWL exists in two versions: The version recommended by W3C in 2004 (often known as OWL 1), and The OWL 2 Web Ontology Language (OWL 2), which became recommended in 2009. OWL 2 is an extension and revision of OWL 1, and is backward compatible for all intents and purposes [**?**].

Both versions of OWL features three sublanguages/profiles[1]:

- OWL 1, with complexity in increasing order (all quoted from OWL Features [**?**]):
    1. OWL Lite (OWLLite): Supports classification hierarchy and simple constraints (e.g. only cardinality values of 0 and 1).
    2. OWL Description Logics (OWL DL): Maximum expressiveness while retaining computational completeness and decidability.
    3. OWL Full (OWLFull): Maximum expressiveness and the full syntactic freedom of RDF, but with no computational guarantees.
- OWL 2 (all quoted from OWL 2 Profiles [**?**]):
    1. OWL Existential Language (OWL EL): Designed to be used with ontologies that contain very large numbers of properties and/or classes.
    2. OWL Query Language (OWL QL): Aimed at applications that use very large volumes of instance data, and where query answering is the most important reasoning task.

[1] This might be wrong: `http://www.w3.org/TR/2004/REC-owl-features-20040210/#s1.3` states that OWL 1 has three sublanguages, while `http://www.w3.org/TR/2009/REC-owl2-new-features-20091027/#Backward_Compatibility` claims that it only has one. But for the purposes of this thesis, we work with three sublanguages.

3. OWL Rule Language (OWL RL): Aimed at applications that require scalable reasoning without sacrificing too much expressive power.

To go through all differences between OWL 1 and OWL 2 would be beyond the scope of this thesis, but suffice to say is that OWL 2 is designed to be backward compatible with OWL 1, and the sublanguages OWL provides as a whole increases the reasoning capabilities of SW.

2.1.4 Linked Data (LD)

A cornerstone of RDF is that all identifications (that is, except BNs) are IRIs. In this way, machines can browse the web for relevant resources, much like you browse the web through hyperlinks. This design feature makes RDF adhere to LD, which is a term that refers to a set of best practices for publishing and connecting structured data on the web [**?**].

Tim Berners-Lee have in his article about LD[2] outlined four "rules" for publishing data on WWW:

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URIs, so that they can discover more things.

These have become known as the "Linked Data principles", and provide a basic recipe for publishing and connecting data using the infrastructure of the WWW while adhering to its architecture and standards [**?**].

LD are relient on two web-technologies, namely IRIs and Hypertext Transfer Protocol (HTTP). Using the two of them you can fetch any resource adressed by an IRI that uses the http-scheme. When combining this with RDF, LD builds on the general architecture of the Web [**?**].

The Web of Data can therefore be seen as an additional layer that is tightly interwoven with the classic document Web and has many of the same properties [**?**]:

- The Web of Data is generic and can contain any type of data.
- Anyone can publish data to the Web of Data.
- Data publishers are not constrained in choice of vocabularies with which to represent data.

[2] `http://www.w3.org/DesignIssues/LinkedData.html`

- Entities are connected by RDF links, creating a global data graph that spans data sources and enables the discovery of new data sources.

2.1.4.1 Linked Open Data (LOD)

Based on the notion of LD, there's a movement to publish data on WWW as LOD. Especially toward governmental institutions there's now an increasing trend of opening data[3].

To encourage this trend, Tim Berners-Lee published a star rating system. On a scale from one to five stars, it rates how well the given dataset is in becomming open. It's incremental, meaning that the dataset needs to be have one star before it can be given two. One star is given if your data is available on WWW with an open license. Two stars means that your data is available in machine-readable structure, and is valid for another star if the structure is a non-proprietary format (e.g. Comma-Separated Values (CSV) instead of Excel). Four stars are given if your the data is identified by using open standards from W3C (e.g. RDF and SPARQL). The last star means that your data also link to other people's data, in order to provide context [**?**].

Figure 2.4 shows the linking open data cloud diagram. It illustrates to some extent the magnitude of data that are linked as of yet[4].

2.1.4.2 URL VS URI VS IRI

Throughout this thesis you'll read the terms Unified Resource Locators (URLs), URIs, and IRIs being used interchangeably. The author strive to use IRI as it's supported by RDF, but in some cases it's more appropriate to use the others because of the texts they reference.

URLs and URIs are the most used terms, as the former denotes derefencible resources on WWW, while the latter is a generalization that can denote anything, even resources not on WWW. But URIs are limited to the character-encoding scheme American Standard Code for Information Interchange (ASCII), and as such IRI has been introduced to solve this problem.

Figure 2.5 explaines the parts of URI. The explanation is equally valid for URL and IRI.

2.1.5 Serializations

RDF in itself offers no serialization of the graph it represents. But there are many serializations available, and more are coming as of this writing.

[3]Examples of this are platforms such as UK's initiative to open governmental data (`http://data.gov.uk/`), US's approach of the same (`http://www.data.gov/`), and Norway's parliament opening of its databases through Stortingets datatjeneste (`http://data.stortinget.no/`). You also have other non-governmental organizations, such as Parliamentary Monitoring Organizations (PMOs).

[4]Well, as of 19-September-2011, when the diagram was last updated.

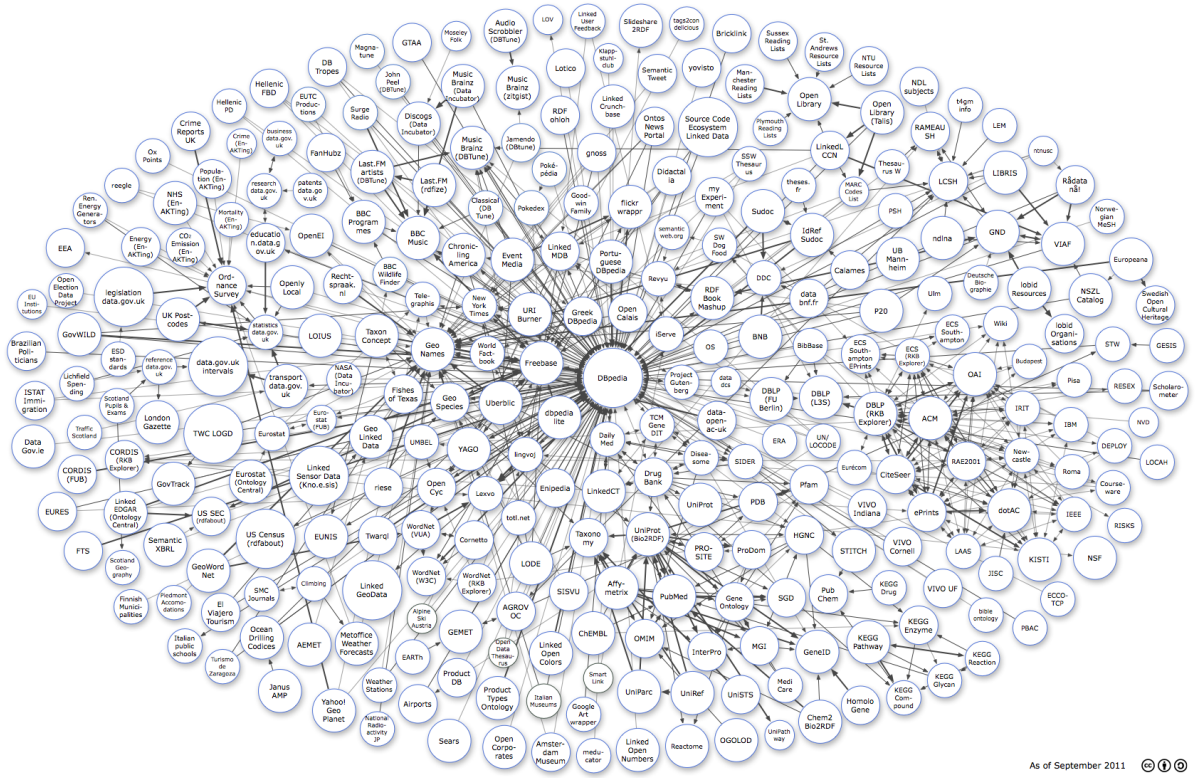

Figure 2.4: Linking Open Data cloud diagram, by Richard Cyganiak and Anja Jentzsch. `http://lod-cloud.net/`

URI have the form $scheme : [//authority]path[?query][\#fragment]$, where the parts in brackets are optional.

- **scheme:** The scheme classify the type of URI, and may also provide additional information on how to handle URIs in applications.
- **authority:** An authority is the provider of content, and may provide user and port details (e.g. arne@semanticweb.com, semanticweb.com:80).
- **path:** The path is the main part of many URIs, though it is possible to use empty paths, e.g., in email adresses. Paths can be organized hierarchically using / as separator.
- **query:** The query can be recognized with the preceding ?, and are typically used for providing parameters.
- **fragement:** Fragments provide an additional level of identifying resources, and are recognized by the preceding #.

Figure 2.5: Shortened explanation on URI given by Hitzler et.al. [**?**, p. 23].

There are some considerations to take when choosing a serialization for a given project. One consideration is the ease for humans to read the syntax, which is very useful if you want to verify how your data is related. Another is the availability of tools to process the serialization. RDF/XML, for example, is based on Extensible Markup Language (XML), and as such there are exists many tools that can deserialize it. Terse RDF Triple Language (Turtle) on the other

hand is specific for RDF, and may not be as easy to deserialize. But most will agree that the latter is much easier to read and understand than the former.

2.1.5.1 RDF/XML

RDF/XML has been recommended by W3C to represent RDF since the beginning [**?**, sec. 2.2.4]. As the name suggests, RDF/XML is based on the markup language XML. XML may not be as humanly accessible as some of the other serializations, but it is the most commonly used, probably because of the readily available software to process XML-documents.

XML is tree-based, which means some consideration when we serialize graphs. Each statement will have the subject as the root, followed by the predicate, and then the object. As an example of this we have listing 2.1, which shows a serialization of figure 2.2.

Listing 2.1: Serialization of figure 2.2 into RDF/XML.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <rdf:RDF xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3           xmlns:foaf="http://xmlns.com/foaf/0.1/">
4
5  <rdf:Description rdf:about="http://example.org/Arne">
6    <foaf:knows>
7      <rdf:Description rdf:about="http://example.org/Kjetil">
8      </rdf:Description>
9    </foaf:knows>
10   <foaf:familyName>Hassel</foaf:familyName>
11 </rdf:Description>
12
13 </rdf:RDF>
```

Another reason for XML being chosen as the default serialization was that it was readily available at the time RDF was being standardized. Figure 2.6 shows a timeline of the development of XML and SW.

Listing 2.1 shows that we have namespaces in XML through the attribute rdf:xmlns. But we cannot use namespaces in values given to attributes (i.e. we have to write rdf:about="http://example.org/Arne" instead of rdf:about="ex:Arne"). This adds to the notion that XML-documents are bigger than what we need to serialize RDF.

2.1.5.2 Terse RDF Triple Language (Turtle)

Turtle defines a textual syntax for RDF that allows RDF graphs to be completely written in compact and natural text form [**?**]. The latest version was submitted as a W3C Team Submission 28 March 2011. Listing 2.2 shows the serialized form of figure 2.2.

We see from the example that IRIs are written with angular brackets, literals with quotation marks, and statements ends with either a semicolon or a period. The usage of semicolon is a

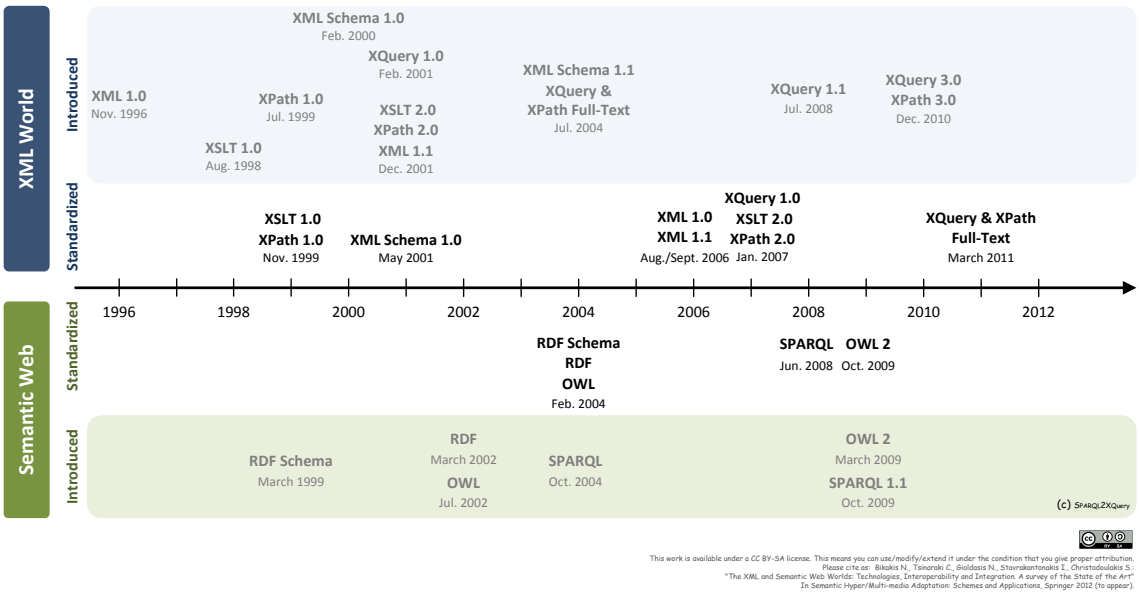

Figure 2.6: XML and Semantic Web W3C Standards Timeline.

Listing 2.2: Serialization of figure 2.2 into Turtle.

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:Arne foaf:knows ex:Kjetil ;
    foaf:familyName "Hassel" .
```

syntactic sugar, and enables writing the following triples without their subject, as they reuse the subject in the first statement. We can also reuse the subject and the predicate in a statement by using the comma, in essence writing a list.

The syntax @prefix is also used in the listing. This allows us to introduce namespaces, and abbreviate IRIs by prefixing them (e.g. http://example.org/Arne → ex:Arne). We also have the term @base, which also enables us to abbreviate IRIs, by writing the suffix in angular brackets (e.g. @base <http://example.org/> → <Arne>).

Turtle also supports BNs by wrapping the statements in square brackets. Listing 2.3 shows all of this syntaxes in use by serialize figure 2.3.

Listing 2.3: Serialization of figure 2.3 into Turtle.

```
@base <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<Arne> foaf:knows [
    foaf:nick "Bjarne" , "Buddy" .
]
```

There's also syntactic sugar for writing collections. This is done by enveloping the resources as

a comma-separated list in parantheses. Lastly, Turtle abbreviates common datatypes, e.g. the number minus five can be written *42*, instead of "42"^^<http://www.w3.org/2001/XMLSchema#integer>, and the boolean true can be written *true* instead of "true"^^<http://www.w3.org/2001/XMLSchema#boolean>.

Turtle has become popular amongst the academic circles of SW, as it's a valuable educational tool because of it's simplicity and readibility.

2.1.5.3 Notation3 (N3)

N3 is often presented as a compact and readable alternative to RDF/XML [**?**], but the syntax supports greater flexibility than the confinements of RDF (e.g. support for calculated entailment with "built-in" functions [**?**]).

It dates back to 1998 [**?**, p. 25], and currently holds status as a Team Submission at W3C, last updated 28 March 2011. Figure 2.2 is serialized as N3 in listing 2.4.

Listing 2.4: Serialization of figure 2.2 into N3.

```
@prefix ex: <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:Arne foaf:knows ex:Kjetil ;
    foaf:familyName "Hassel" .
```

N3 shares a lot of the syntax of Turtle, but is an extension in the regard that it has extra syntax (e.g. @keywords, @forAll, @forSome) [**?**, sec. 9].

2.1.5.4 N-Triples

N-Triples was designed to be a fixed subset of N3 [**?**, sec. 3]. It's also a subset of Turtle, in that Turtle adds syntax to N-Triples [**?**, sec. 8]. Serialization of figure 2.2 is given in listing 2.5.

Listing 2.5: Serialization of figure 2.2 into N-Triples.

```
<http://example.org/Arne> <http://xmlns.com/foaf/0.1/knows> <http://example
    .org/Kjetil> .
<http://example.org/Arne> <http://xmlns.com/foaf/0.1/familyName> "Hassel" .
```

One way of looking at N-Triples is to see it as Turtle without the syntactic sugar.

2.1.5.5 RDF JSON

RDF JSON was one of the earliest attempts to make a serialization of RDF in JavaScript Object Notation (JSON). It's designed as part of the Talis Platform[5], and is a simple serialization of RDF into JSON. Figure 2.2 is serialized into RDF JSON in listing 2.6.

Listing 2.6: Serialization of figure 2.2 into RDF JSON.

```
{
    "http://example.org/Arne": {
        "http://xmlns.com/foaf/0.1/knows": [ {
            "value": "http://example.org/Kjetil",
            "type": "uri"
        },
        "http://xmlns.com/foaf/0.1/familyName": [ {
            "value": "Hassel",
            "type": "literal"
        }
    }
}
```

All triples have the form "S" : "P": [O] , where S is the subject, P is the predicate, and O is a JSON object with the following keys:

- type, required: either "uri", "literal" or "bnode"
- value, required: the lexical value of the object
- lang, optional: the language of the literal
- datatype, optional: the datatype of the literal

2.1.5.6 JavaSript Object Notation for Linked Data (JSON-LD)

JSON-LD is the newest serialization to be included by W3C. It became a working draft 12 July 2012, after being in the works for about a year by the JSON for Linking Data Community Group (JSON-LD CG)[6]. It has been included in the work of the RDF Working Group (RDF WG) in hope that it will become a Recommendation that will be useful to the broader developer community[7].

JSON-LD CG has from the start worked with the concern that RDF may be to complex for the JSON-community[8], and as such has embraced LD rather than RDF. That being said, it is a goal that JSON-LD will serialize a RDF graph, *if* that is what the developer want to do. This

[5] http://docs.api.talis.com/platform-api/output-types/rdf-json
[6] http://json-ld.org/, http://www.w3.org/community/json-ld/
[7] http://www.w3.org/blog/SW/2011/09/13/the-state-of-rdf-and-json/
[8] Topic: Formal Definition of Linked Data at http://json-ld.org/minutes/2011-07-04/

is reflected in the current working draft, in that subjects, predicates and objects "*SHOULD* be labeled with an IRI".

Another design goal of JSON-LD is simplicity, meaning that developers only need to know JSON and two keywords (i.e. @context and @id) to use the basic functionality of JSON-LD [**?**, sec. 2]. So how do we use these keywords? Lets look at two examples in listings 2.7 and 2.8, which serialize figures 2.2 and 2.3 respectively.

Listing 2.7: Serialization of figure 2.2 into JSON-LD.

```
1  {
2      "@context": {
3          "ex": "http://example.org/",
4          "foaf": "http://xmlns.com/foaf/0.1/"
5      },
6      "@id": "ex:Arne",
7      "foaf:knows": "ex:Kjetil",
8      "foaf:familyName": "Hassel"
9  }
```

Listing 2.8: Serialization of figure 2.3 into JSON-LD.

```
1  {
2      "@context": {
3          "ex": "http://example.org/",
4          "foaf": "http://xmlns.com/foaf/0.1/"
5      },
6      "@id": "ex:Arne",
7      "foaf:knows": {
8          "foaf:nick": [ "Bjarne", "Buddy" ]
9      }
10 }
```

In listing 2.7 we see that prefixing namespaces are featured in line 3 and 4. We also see that the subject are defined by using the @id-property. The absence of @id-property though creates a blank node, as shown in listing 2.8.

Another design goal of JSON-LD is to provide a mechanism that allow developers to specify context in a way that is out-of-band. The rationale behind this is to allow organizations that already have deployed large JSON-based infrastructure to add meaning to their JSON documents that is not disruptive to their day-to-day operations [**?**]. In practice this will work by having two JSON documents, one being the original JSON document, which isn't linked, and another that provide rules as to how terms should be transformed into IRIs. Listing 2.9 shows how a serialization of figure 2.1 could be transformed into the serialization of 2.2.

Listing 2.9: Framing in JSON-LD.

```
// A non-LD JSON object
{
    "Arne": {
        "knows": "Kjetil",
        "lastname": "Hassel"
    }
}
// A JSON-LD object designed to transform the object above into a JSON-LD
    compliant object
{
    "@context": {
        "ex": "http://example.org/",
        "foaf": "http://xmlns.com/foaf/0.1/",
        "Arne": {
            "@id": "ex:Arne"
        },
        "Kjetil": {
            "@id": "ex:Kjetil"
        },
        "knows": "foaf:knows",
        "lastname": "foaf:familyName"
    }
}
```

2.1.5.7 Resource Description Framework in Attributes (RDFa)

RDFa is another serialization that recently got promoted in the W3C-system. As of June 7th 2012 it's a W3C Recommendation, and offers a range of documents (the RDFa Primer[9], RDFa Core[10], RDFa Lite[11], XHTML+RDFa 1.1[12], and HTML5+RDFa 1.1[13]).

RDFa makes it possible to embed metadata in markup-languages (e.g. Hypertext Markup Language (HTML)), so as to make it easier for computers to extract important information. This is in response to the fact that some semantics may not be specific enough. Take the title-tags in HTML, H1-H6. Good practices suggest only using H1 one time, so that it only specifies the most important title for the page. But even so, what does the H1-tag specify title for? Is it the page as a whole, or is it the specific article on that page. With RDFa you can specify this in a way that doesn't leave any doubt.

The reasoning is that by making use of independently created vocabularies, the quality of meta-data will increase. And by tying it into RDF, you can increase the overall knowledge of WWW.

RDFa has a syntax much to big to describe in detail here, but lets look at an example, by serializing figure 2.2 into a fracture of HTML, given in listing 2.10.

[9] http://www.w3.org/TR/rdfa-primer/
[10] http://www.w3.org/TR/rdfa-core/
[11] http://www.w3.org/TR/rdfa-lite/
[12] http://www.w3.org/TR/xhtml-rdfa/
[13] http://www.w3.org/TR/rdfa-in-html/

Listing 2.10: Serialization of figure 2.2 in RDFa.

```
<div
  vocab="http://example.org/"
  prefix="foaf: http://xmlns.com/foaf/0.1/"
  about="Arne">Arne knows
  <span
    property="foaf:knows"
    resource="Kjetil">Kjetil</span>
  and has last name <span
    property="foaf:familyName">
    Hassel</span>.</div>
```

Listing 2.10 shows us the use of the attributes vocab, prefix, about, property, and resource:

- Vocab defines the usage of a single vocabulary for the nested terms, and may be overridden by setting vocab with another IRI.
- Prefix allows us to introduce prefixes in case we want to mix in more vocabularies.
- About defines the subject in a triple
- Property defines the predicate in a triple
- Resource may define the object and the subject, depending on context

2.1.6 Querying

An important feature of structured data is the possibility of querying it. You could have the users scour the RDF in tools like a SW or RDF browser, but this can be a tedious task, and very inefficient for a machine. To query RDF we need a query language that recognizes RDF as the fundamental syntax [**?**, p. 192] (or rather, as the fundamental model).

2.1.6.1 SPARQL Protocol and RDF Query Language (SPARQL)

SPARQL is the answer to the need for a query language. It exists as version 1.0, which became a W3C Recommendation January 15th 2008, and as version 1.1, which is a working draft, last updated January 5th 2012. Version 1.1 builds upon version 1.0, and sports features such as [**?**]:

- The query forms SELECT, ASK, CONSTRUCT, and DESCRIBE
- Grouping, ordering, and limitation of results fetched
- Several shortened query forms
- Aggregation

- Subqueries
- Negation
- Expressions in the SELECT clause and Property Paths
- Assignment
- A large list of functions and operators

As the most powerful version, we'll use version 1.1 as the basis for this thesis, and will be the version we refer to when referring to SPARQL.

There are four fundamental forms of read-queries in SPARQL, namely SELECT, ASK, CONSTRUCT, and DESCRIBE. The two latter returns new graphs, that can be used as basis for additional queries and manipulations (e.g. merging with other graphs).

The SELECT form enables us to query for variables, and return them in a tabular form. We can project a specific list of variables we want returned, or just select all variables by using the asterix sign.

Listing 2.11 shows a very simple example of a SELECT-query. If we use that query against the model in figure 2.2, well get the table 2.1 as a result.

Listing 2.11: An example of the SELECT form in SPARQL

```
1 SELECT *
2 WHERE { ?subject ?predicate ?object }
```

?subject	?predicate	?object
http://example.org/Arne	http://xmlns.com/foaf/0.1/knows	http://example.org/Kjetil
http://example.org/Arne	http://xmlns.com/foaf/0.1/familyName	"Hassel"

Table 2.1: Result from using query in listing 2.11 on the model in figure 2.2

As we see from table 2.1, the query lists all triples we know in the model.

The ASK form enables us to verify whether or not certain query pattern are true or not. We could use it to ask if we know from the model in figure 2.2 whether or not there are an entity which has a given name "Arne". Listing 2.12 shows how this is done.

Listing 2.12: An example of the ASK form in SPARQL

```
1 @prefix foaf: <http://xmlns.com/foaf/0.1/>
2 ASK { ?x foaf:givenName "Arne" }
```

In our case the result would be *false*.

Listing 2.13: An example of the CONSTRUCT form in SPARQL

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
CONSTRUCT { ?x foaf:givenName "Arne" }
WHERE { ?x foaf:familyName "Hassel" }
```

The CONSTRUCT form enables us to derive a graph derived from other graphs. Lets look at another example in listing 2.13.

Now, if we were to run the ASK-query in listing 2.12 against the new graph, we would get the result *true*. And if we ran the SELECT-query in listing 2.11, we would get the result in table 2.2.

?subject	?predicate	?object
http://example.org/Arne	http://xmlns.com/foaf/0.1/givenName	“Arne”

Table 2.2: Result from using query in listing 2.11 on the graph resulting from the query in listing 2.13 begin executed on the model in figure 2.2.

The DESCRIBE form results in a single RDF graph. It differs from the CONSTRUCT form in that we doesn't specify which triples we want the new graph to consist of, but rather that the SPARQL query processor determines which triples that are relevant. Which triples are relevant depend on the data available in the graph(s) queried, but takes basis in the resources identified in the query pattern.

Lets look at the query in listing 2.14, which we apply to the models in figures 2.2 and 2.3, which we've assigned to IRIs `http://example.org/GraphA` and `http://example.org/GraphB` respectively. The result could be something like the serialization shown in listing 2.15.

Listing 2.14: An example of the DESCRIBE form in SPARQL

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>
@prefix ex: <http://example.org/>
CONSTRUCT ?y
FROM <http://example.org/GraphA>
FROM NAMED <http://example.org/GraphB>
WHERE { ?x foaf:knows ?y }
```

Listing 2.15: A possible serialization of the result from the query in listing 2.14

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
[ foaf:nick "Bjarne" , "Buddy" . ]
```

The resulting graph has two triples, namely the one concerning the entity which we known has the nicks “Bjarne” and “Buddy”. As there are no triples where `http://example.org/Kjetil` acts as the subject, we can't describe anything.

We've introduced the token FROM in the query. This syntax allows us to specify which RDF Datasets we wish to query. This syntax is optionable, as the query processor will use the default graph if nothing is specified. There can be one default graph, whose IRI we override if we specify FROM without NAMED. A query can take any number (or none) of named graphs, but don't need a default graph if we have one or more named graphs.

SPARQL has a great number of features, and we can't describe them all here[14]. But suffice to say, SPARQL is a powerful language that enables us to ask a variety of questions regarding our data.

2.1.6.2 SPARQL Update Language

The SPARQL 1.1 specification is part of a set of documents, which comprises of ten documents. One of these is the document regarding SPARQL Update Language. It introduces an extension of the SPARQL syntax that allow us to update RDF datasets. The tokens are divided into two groups, Graph Update and Graph Management. The former consists of INSERT DATA, DELETE DATA, DELETE/INSERT (with the shortcut form DELETE WHERE), LOAD, and CLEAR. The latter consists of CREATE, DROP, COPY, MOVE, and ADD.

We won't go into detail, but SPARQL Update Language delivers a great variety of terms that allows us to manipulate our graphs with SPARQL.

2.1.7 Entailment

An important feature of RDF is the ability to infer knowledge from the excisting knowledge, i.e. form or entail new conclusions. This is referred to as entailment. There are multiple forms of entailments in RDF, and it supports one form "out-of-the-box". The document "RDF Semantics"[15] gives details about entailment for RDF, RDFS, and D-entailment.

Other regimes are the OWL Direct Semantics[16], which covers OWL DL, OWL EL, and OWL QL. There's also the idea of Rule Interchange Format (RIF), which outlines a core syntax for exchanging rules. The idea is to support multiple rule language, instead of the specific entailment regimes.

As entailment didn't become a part of the framework implemented as part of this thesis, we won't go into greater detail at this point. We'll return to entailment in section 6.3 as part of the discussion.

[14] The SPARQL 1.1 specification numbers almost 100 pages as of this writing
[15] `http://www.w3.org/TR/rdf-mt/`
[16] `http://www.w3.org/TR/owl2-direct-semantics/`

2.2 Javascript (JS)

JS begins its life in 1995, then named Mocha, birthed by Brendan Eich at Netscape [**?**, **?**]. It then got rebranded as LiveScript, and later on Javascript when Netscape and Sun got together. When the standard was written, it was named ECMAScript, but everyone knows it as Javascript. It quickly gained traction for its easy inclusion into webpages, but was long rediculed and misunderstood by developers [**?**].

Douglas Crockford states in his article "JavaScript: The World's Most Misunderstood Programming Language" ten reasons for the confusion centering JS, given in the list below. Luckily there has been some changes to the list since its conception in 2001.

1. The Name
2. Lisp in C's clothing
3. Typecasting
4. Moving Target
5. Design Errors
6. Lousy Implementations
7. Bad Books
8. Substandard Standard
9. Amateurs
10. Object-Oriented

Point 1-5 is quite valid yet[17], but can be remedied by good and educational resources for learning JS[18]

Point 6 is (mostly[19]) not valid anymore. If the community learned anything from the browser wars, it was to work with the community through the process of standards. Ecma Internationals effort to codify the de facto standard amongst the browsers has proved increasingly successful, groups such as W3Cs HTML Working Group and The Web Hypertext Application Technology Working Group (WHATWG) drives the production of standards, and great efforts are made to increase efficiency amongst JS-engines. Another testimony to the fact that implementations are

[17]Design issues in JS has given rise to many frustratring moments, creating momentum for websites such as `http://wtfjs.com/`, which delivers examples of "weird code".

[18]In this regard, `http://dailyjs.com/` offers a variety of good resources for learning JS, specifically its articles tagged with #beginner (`http://dailyjs.com/tags.html#beginner`).

[19]Considering the slow pace of browser-update in some communities, e.g. usage of Internet Explorer version 6 (IE6) in China, lousy implementations is a thing that is becomming a thing of the past.

increasingly popular are the efforts to use JS as a programming language outside the browser (described in section 2.2.7).

Point 7 depends on your view of good books, and although there's much left to desire, there are some good books out there[20]. But more importantly, there are several efforts to deliver resources of high quality to educate developers in JS. These resources are increasingly - perhaps fittingly - web-based. There's also an increase of interest on conferences that target developers[21].

Point 8 is left to be discussed (the author haven't read and analyzed the 440 pages that ECMAScript version 3 and 5 consists off), but the implementation of the standards seem to suggest that this point is not so valid anymore.

JS is increasingly becomming part of the professional world, adaptations into conferences being one of the arguments suggesting this trend. You also have examples of major companies supporting and/or developing JS-libraries[22]. This would suggest that point 9 is not the case anymore[23].

Point 10 is still valid, as it can be hard for developers trained in conventional object-oriented languages like Java and C#. Again, as with point 1-5, this is remedied by proper, educational resources, that developers can turn to when puzzled by the intricacies of JS.

JS may be a greatly misunderstood language still, but it seems to have a lot going for it. The fact that it is the de facto programming language for the web puts it into a position worthy of respect, and should be regarded as a resource which can be used for many great things.

In this thesis we will differentiate between the implementation of JS and the specification ECMA-262 5.1 Edition (ECMA5). ECMA5 will be specified when there's functionality that haven't been implemented in major browser yet.

2.2.1 Object-Oriented

JS is fundamentally Object-Oriented (OO) as objects are its fundamental datatype [**?**, p. 115]. It treats objects different than many other programming languages though, as it doesn't have classes and class-oriented inheritance. There are fundamentally two ways of building up object

[20] Which include, in the authors view:

- JavaScript: The Definitive Guide, 6th edition, by David Flanagan (O'Reilly Media)
- JavaSCript: The Good Parts, by Douglas Crockford (O'Reilly Media)

[21] Notably, in Norway you have Web Rebels (`http://webrebels.org/`), and JS has its own session on Norwegian Developers Conference (NDC), with somewhat above 10% of the talks concerning JS.

[22] One example being jQuery, which is shipped with Microsoft's Visual Studio, another being AngularJS, which is an MIT-licensed Model-view-controller (MVC)-framework developed by Google.

[23] That being said, percentage-wise it's probable that JS is still written by more amateurs than professional developers. This is not a bad thing though, as it expresses the power of adaptation that JS features, and may be a gateway for developers-to-be. Also, lets not forget that the word amateur means "lover of", and love of computer technologies is something to be embraced.

systems, namely by inheritance (explained in section 2.2.1.1 and by aggregation (explained in section 2.2.1.2) [**?**].

Another design feature is its support of the functional programming style, by treating objects as first-class functions. This feature is explained thoroughly in section 2.2.1.3.

The level of object-orientation in JS is shown in that even literals (i.e. all primitive values except Undefined and Null) can be treated as objects. They are, however, immutable, and doesn't share the dynamic properties that "normal" objects in JS do. JS handles this by wrapping the values into their respectively object-type (e.g. String, Number, and Boolean). An example showing this is shown in listing 2.16.

Listing 2.16: Use of literals in JS

```
var stringObject = new String('foo');
console.log(stringObject.length); // logs 3
var stringLiteral = 'foo';
console.log(stringLiteral.length); // logs 3
```

Other objects that are somewhat different from the norm is the Array- and Math-object, the former representing a list of values and the latter sporting a set of static methods.

An aspect of this that's worth mentioning is the fact that JS is class-less. That means that objects in JS don't need classes to be instantiated. It is possible to emulate classes in JS though, as it helps us use class-depended features (e.g. some Software Design patterns (SDPs)), and an example is shown in listing 2.17.

Listing 2.17: Emulation of classes in JS

```
var MyClass = function () {
    this.myProperty = 42;
    this.myMethod = function (value) {
        return value + this.myProperty;
    };
};

var myObject = new MyClass();
console.log(myObject.myMethod(1295)); // logs 1337
```

2.2.1.1 Prototypical Inheritance

At the heart of all object-handling in JS is Object. All objects inherit this object if nothing else is specified, and it is there we find the default properties and methods that are shared by all objects. We can manipulate which object we want our objects to inherit, and as such can create a hierarchy of objects. Listing 2.18 show some examples of inheritance. In it we see how we can initiate objects, and how we can assign them to inherit other objects.

Listing 2.18: Usage of prototype in JS

```
var objectA = {};
console.log(objectA); // logs Object
console.log(objectA.__proto__); // logs Object
Object.prop = 42;
console.log(objectA.prop); // logs 42

var objectB = new Object(),
    objectC = Object.create(objectB);
console.log(objectC.prop); // logs 42
```

The simple secret behind prototypical inheritance is that all objects have the property *__proto__*. When a property or method is called, JS will search for the called element by traversing the objects' properties, and if not found, it will continue with the prototype. We can visualize the structure in listing 2.18 as a tree, and have done so in figure 2.7.

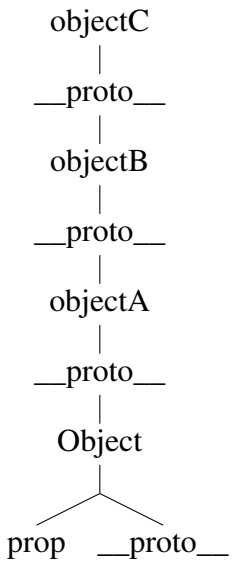

Figure 2.7: Object inheritance created in listing 2.18 visualized as a tree.

So when we call objectB.prop, JS will see if objectB has that property. As it hasn't it will continue to its prototype, which is objectA. As there is no objectA.prop, it will look for the prototype, which contains a link to Object. Now, as Object has the property prop, JS will return its value, which is 42 in our case.

A last note is that Object also have the property __proto__. This can also be manipulated, but JS will take care so that we don't run into an infinite loop when looking for properties that doesn't exist (it's also considered a bad practice (i.e. an anti-pattern) to manipulate the prototype of Object).

2.2.1.2 Dynamic Properties

All mutable objects in JS can be manipulated at run-time. This we also see in listing 2.18, as we add the property "prop" in line 4. Objects are basicly as containers for key-value entities, where the key is a string. In this regard, objects in JS can be regarded as maps, or dictionaries.

In addition, we can at any time manipulate existing properties by replace its values or delete the key alltogether. We can also manipulate objects that are prototyped, and the objects that inherit will also be affected. How this works is that JS creates a reference in memory for variables that are set as objects. If those variables where to be set to other variables, the reference would be copied, not the values contained within.

A note on mutability and immutability: JS differentiate between primitive values and object. The former are immutable, while the latter is mutable. ECMA5 offers three new functions that change this feature, namely Object.seal, Object.freeze, and Object.preventExtensions (with the responding Object.isSealed, Object.isFrozen, and Object.isExtensible to test whether or not these are set) [**?**, p. 114-115]. Explaining how these functions are outside the scope of this thesis, but suffice to say is that ECMA5 adds some spice to the mutable properties of JS-objects.

2.2.1.3 Functional Features

All functions are treated as first-class objects, and as such can be manipulated as any other object. It can also be passed around as variables, and this opens for some nifty features. By passing a function as a parameter, we can call that function whenever we want, e.g. after we've loaded a set of resource. This asynchronous feature is explained in depth in section 2.2.5.

Functions can be instantiated in many ways, as shown in listing 2.19. A function consists of three elements [**?**, p. 164]:

1. Name: An identifier that names the function (optional in function definition expressions).
2. Parameter(s): A pair of parentheses around a comma-separated list of zero or more identifiers.
3. Body: A pair of curly braces with zero or more JS-statements inside.

All types in listing 2.19 support these requirements, albeit a little differently. Line 1 shows a named function, while the other two are anonymous. Anonymous functions are called through their reference, i.e. the variables they are set to. Named functions is referrable by their names, if not they are set to a variable, in which case it will be referrable by the variable (line 10 shows what happens if you call the function by its name when its set to a variable).

Functions of the types listed in line 1-3 can be used as constructors for new objects, while the one in line 4 can be used as a prototype. A simple example of this is shown in listing 2.20. It introduces the use of "this", which will be explained in section 2.2.2.

Listing 2.19: Instanciating functions in JS

```
function functionA (x) { return x; };
var functionB = function (x) { return x; },
    functionC = function functionD (x) { return x; },
    functionE = new Function("x", "return x;");

console.log(functionA(42), // logs 42
            functionB(42), // logs 42
            functionC(42), // logs 42
            functionD(42), // throws ReferenceError: functionD is not
                defined
            functionE(42)); // logs 42
```

Listing 2.20: A simple object in JS

```
function ObjectA (x) {
    this.x = x;
    this.methodA = function (y) {
        return this.x + y;
    };
}

var A = new ObjectA(1300);
console.log(A.methodA(37)); // logs 1337
```

2.2.2 Scope in JS

The way JS handles the scope may be confusing to developers coming from class-oriented programming languages. JS doesn't contain syntax such as private of protected for use with variables, but it supports private variables for objects. It does so in the way it handles the context functions are part of (e.g. the scope).

Functions in JS can be nested within other functions, and they have access to any variables that are in scope where they are defined. This means that JS-functions are closures, and it enables important and powerful programming techniques [**?**].

If a variable isn't set as a property in an object, it will be a part of the global object. The global object in JS depends on which environment it's run in, but in most browsers it's represented by the object "window". This has some consequences, like the fact that usage of the syntax-element "var" is optional; it will become a key-value entity in the scope in which it's declared, which is the global object if nothing else is specified. This is examplified in listing 2.21.

2.2.2.1 Closure

Lets review a simple example of closure, given in listing 2.22. In this example we have two functions, one which works as a constructor, and another that merely calls a function it has been given as parameter. When we pass a.getValue to functionA, JS also include the context which

Listing 2.21: Examples of scope in JS

```
var x = 42;
y = 42;
window.z = 42;

console.log(x, y, z); // logs 42 42 42
```

that method runs in, in effect creating a closure.

Listing 2.22: A simple example of closure in JS

```
var ObjectA = function (val) {
            this.val = val;
        this.getValue = function () {
            return this.val;
        }
    },
    functionA = function (getFunc) {
        return getFunc();
    };

var a = new ObjectA(42);
console.log(functionA(a.getValue)); // logs 42
```

This effect is increasingly used in JS-libraries, and is getting a lot of appraise from the community. But it's also a headache for many aspiring JS-developers, as it may be a bit difficult to wrap your head around (and use correctly). Let's look another example of what may go wrong, given in listing 2.23. In this example we try to access this.val inside functionAA. But as functionAA is not part of the scope of functionA, and thereby not being a part of the closure given to functionB, we fall back to calling on the global object. Since the global object doesn't have a property named val, it will return "undefined".

Listing 2.23: An example of code gone wrong because of faulty handling of closure

```
var functionA = function (val, func) {
        this.val = val;
        function functionAA () {
            return this.val;
        }
        return func(functionAA);
    },
    function B = function (func) {
        return func();
    };

console.log(functionA(42, functionB)); //logs undefined
```

2.2.3 Static functions

JS supports static functions in that all functions are treated as objects, and by extension can be extended with methods. Listing 2.24 illustrates an example of this.

Listing 2.24: An example of static functions in JS

```
1 var funcA = function (val) { this.val = val; },
2     objA = new funcA(1337);
3 funcA.funcB = function () { return 42; }
4 funcA.funcC = function () { return this.val; }
5 console.log(funcA.funcB()); // logs 42
6 console.log(objA.funcC()); // throws TypeError
7 console.log(funcA.funcC.call(objA); // logs 1337
```

Note that static functions aren't accessible as methods in objects constructed with the parenting functions as constructor. But we can manipulate the scope of the functions by overloading this (with call) to be the object we wish to refer to), as shown on line 7.

2.2.4 JavaScript Object Notation (JSON)

JSON is a lightweight, text-based data interchange format. It's originally based on JS, but is language-independent [**?**]. It was specified by Douglas Crockford in RFC 4627, and enjoys support in most major programming languages.

Needless to say, JS supports JSON by default.

2.2.5 Asynchronous Loading of Resources

Asynchronous loading or resources are common in browsers. Normal HTML documents normally externalize much of its Cascading Style Sheets (CSS) and JS functionality, as dictated by good practices. Those resources are loaded by the browser by default, without too much hassle. But when it comes to making use of the browsers API (i.e. the ones available to JS) to load resources asynchronously, it becomes another game entirely.

2.2.5.1 Same Origin Policy (SOP)

As with many issues, handling external resources are difficult in JS because of security issues. And justifyable so, as JS becomes an increasingly powerful programming language, so are the possibilities to abuse it. Users of WWW are increasingly used to insert personal information, and if we cannot trust owners of webpages to control what's being run on their site, then there would be a lot of issues with trust on the web[24].

[24] A lot can be said about trust on the web, but that is not the purpose of this thesis.

Perhaps the most important security concept within modern browsers is the idea of SOP[25]. Although there is no single SOP governing how browsers implement it, the idea is that resources that don't share the same origin (i.e. having the same scheme, host, and port in the IRI (concepts explained in figure 2.5)) are isolated from each other.

It is possible to circumvent SOP in JS by inserting a script-tag referring to an external file. It's the basis for JSON with padding (JSONP), which allows JSON residing in external files to be loaded during run-time.

2.2.5.2 Content Security Policy (CSP)

Another way of handling security concerning external resources is CSP. CSP is in the works (Working Draft at W3C[26]), and as an incomplete standard it may be prone to changes. But the basic idea is to let developers whitelist external resources. The policy is first and foremost being designed to be part of the HTTP response header, but there is also work on letting it be a part of HEAD-part in a HTML document, as a META-tag.

2.2.5.3 XMLHttpRequest Level 2 (XHR2)

XMLHttpRequest (XHR) has been part of the world of browsers for a while. It was conceived by Microsoft in their work on Microsoft Exchange Server 2000, and was later ported by Mozilla. It was overlooked for quite a while, until AJAX became a trend, as developers understood the power it had to load resources asynchronously (and synchronously, if needed).

XHR2 is a Working Draft as of this writing, but introduces several features requested by the community, allowing cross-domain fetching of resources being one of them. To allow this, it makes use of another standard which is in the making, namely Cross-Origin Resource Sharing (CORS)[27].

This technology is already available in some browsers. But its inherit problem is that it requires domain-owners to add information to their HTTP headers.

Another technology developed to fetch resources across domains are XDomainRequest (XDR). But as it wasn't included in the framework, we've let it be a part of the discussion in section 6.5.

2.2.6 CommonJS (CJS)

CJS is a volunteer-driven project aiming to standardize and implement standards that expand the functionality of JS. It specifications include handling of modules, unit testing, packaging,

[25] `http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy`
[26] `http://www.w3.org/TR/CSP/`
[27] `http://www.w3.org/TR/cors/`

Input/Output (I/O), handling of binary data, and much more. We've included details concerning two of these specifications, as they've been included in the framework.

2.2.6.1 Asynchronous Module Definition (AMD)

AMD is titled Modules/Async/A by CJS[28]. It's overall goal is to provide a solution for modular JS that developers can use today [**?**]. Essentially it makes use of the functions *define* and *require*. The former defines a module, while the latter enables us to load dependencies that the module requires.

AMD allows us to split our functionality into modules and easily load components as they are needed, in run-time. This in turn leeds do a more decoupled codebase, making it easier to make modules reusable. But it may also increase the loading time required, as each module requested fires a HTTP request. Which consequences this has for the framework is further discussed in section 6.6.

2.2.6.2 Promise Pattern

The promise pattern is titled Promises/A by CJS[29]. It's also referred to as Deferred, and works by having an object represent a promise. The object promises that a result will be returned at some time in the future. This can be set up so that when the result is ready, a function is called with the result sent as parameter.

Listing 2.25 shows some examples of the API. A central point of these examples are that the functions passed as parameters to the then-function are called as soon as the promise are resolved, i.e. detached from the order in which they were called in the code.

2.2.7 Server-side implementations

As JS has become an increasingly popular programming language, so has its use outside of the browser. One of these branches is the use of JS for server-side web-applications. As part of this thesis we've only used one such implementation as a run-time environment for our Test-driven development (TDD), but a more in-length discussion of the matter can be found in section 6.4.

2.3 Software Design pattern (SDP)

Patterns were originally conceptualized as an architectural concept by Christopher Alexander, who wrote [**?**, p. x]:

[28] `http://wiki.commonjs.org/wiki/Modules/Async/A`
[29] `http://wiki.commonjs.org/wiki/Promises/A`

Listing 2.25: Examples of the Promise API

```
// When is available as a global variable
var promiseA = When.defer(),
    promiseB = When.defer();
setTimeout(function () { promiseA.resolve(42); }, 2000));
setTimeout(function () { promiseB.resolve(1337); }, 1000));

// Preparing single promises
promiseA.then(function (result) {
    console.log(result); // logs 42 after 2000 milliseconds
});
promiseB.then(function (result) {
    console.log(result); // logs 1337 after 1000 milliseconds
});

// Preparing multiples promises
When.all([ promiseA, promiseB ], function (results) {
    console.log(results); // logs [ 42, 1337  ] after 2000 milliseconds
});
```

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Alexander's work inspired amongst others Kent Beck and Ward Cunningham, who in 1987 presented the report "Using Pattern Languages for Object-Oriented Programs"[30] on OOPSLA-87. In it they outlined the adaptation from Pattern Language to object-oriented programming, and they summarized a system of five patterns that they had successfully used for designing window-based user interfaces.

But SDPs didn't become popularized before the publication of Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (often known as Gang of Four (GoF)) in 1994. They generalize patterns to have four essential elements:

1. Pattern name: A handle which we can use to describe a design problem, its solutions, and consequences in a word or two. A pattern ame is useful as a higher level of abstraction, increases our pattern vocabulary, and eases communication in social contexts.

2. Problem: Each pattern is designed to handle a specific problem, and this part tells us when it's appropriate to use a specific SDP.

3. Solution: This part explains in detail how to solve the given problem by explaining the elements that make up the design, their relationships, reponsibilities, and collaborations.

[30] http://c2.com/doc/oopsla87.html

4. Consequences: All implementations have consequences, and this part tells us what results and trade-offs we may expect from applying the pattern. Consequences may be how the pattern affects a system's flexibility, extensibility, or portability.

In the book they also design a classification scheme that aims to enable developers to refer to families of SDPs. They categorize by a SDPs purpose, which can be either creational, structural, or behavioral. The second categorization is by scope, which can be classes or objects. SDPs that are related to this thesis have been classified in table 2.3.

		Purpose		
		Creational	Structural	Behavioral
Scope	Class			
	Object	Builder (page 32) Singleton (page 37)	Bridge (page 31) Composite (page 33) Decorator (page 34) Facade (page 35) Proxy (page 35)	Strategy (page 39)

Table 2.3: Categorization of SDPs relevant to this thesis, given in the classification scheme proposed by Erich Gamme, et.al. [**?**, p. 10].

At this point we need to make two points clear. The first is that as JS is a class-less programming language, the categorization class might be a bit off. But remember that we can emulate classes in JS, and this allows us to make use of the class-categorized patterns. The other point is that JS doesn't support interfaces. Interfaces can be emulated (at the cost of complexity), but isn't anything more than a construct that enforces that a list of properties is set at run-time. Therefore we have excluded the use of interfaces in this thesis, falling back to merely describing the abstractions of participants, and how they're represented in the code samples.

GoF continues to describe a consistent format for describing SDPs, which including Pattern Name and and Classification, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, and Related Patterns. Using all of these labels takes a lot of pages, and in this thesis we've limited ourselves to a description of the pattern along with a figure and an example in JS.

2.3.1 Bridge

The Bridge pattern "[d]ecouple an abstraction from its implementation so that the two can vary independently" [**?**]. The pattern is actually often used in JS in terms of event handling, using code such as the one in listing 2.26. In that case, the abstraction is that a function is to be called when a specific button is called, the refined abstraction is the actual functions. The implementor on the other hand is a function that takes the id to the button to be handled, and the abstraction that is to be coupled. The concrete implementor is the function handleClick, which configures the setup needed.

We could've implemented another abstraction, namely making sure that whatever was passed as handleClick's first parameter was an object that supported the onclick-property. This way, we could've removed our limitation of sending just strings of ids, e.g. passing the object returned from document.getElementByClass("buttons").

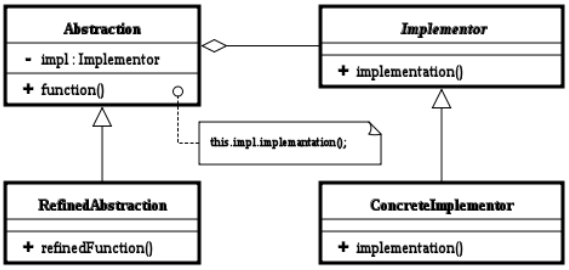

Figure 2.8: Structure of Bridge

Listing 2.26: An example of implementation of Bridge in JS

```
var cancelFunction = function () {
        console.log("Cancel was clicked");
    },
    submitFunction = function () {
        console.log("Submit was clicked");
    },
    handleClick = function (buttonId, func) {
        document.getElementById(buttonId).onclick = function () {
            func();
            return false;
        };
    };
handleClick("CancelButton", cancelFunction); // When clicked, will log "
    Cancel was clicked"
handleClick("SubmitButton", submitFunction); // When clicked, will log "
    Submit was clicked"
```

2.3.2 Builder

The Builder pattern "[s]eparate the construction of a complex object from its representation so that the same construction process can create different representations" [**?**]. A good example of this is the way jQuery allows us to construct DOM elements (listing 2.27).

Listing 2.27: Examples of the Builder pattern in jQuery

```
var paragraph = $("<p>"),
    titleWithText = $("<h1>Our title</h1>"),
    inputWithAttr = $('<input type="password" />');
```

These lines should be very easy to read for developers familiar with HTML, and handles a lot

of logic that is run behind the scene (e.g. the use document.createElement, adding attributes, and text).

Now, lets look at listing 2.28 for our own version of a DOM-builder (a very limited version, e.g. it only support one level of element). We've removed parts of the code, as they unnecessary to understand how the pattern works. We have the DOMCreator (the Director), the DOMBuilder (ConcreteBuilder), and the DOMElement (the Product). The code works in following steps:

1. We pass to DOMCreator the string we wanted parsed.
2. DOMCreator creates an instance of DOMBuilder, and passes along the tag.
3. DOMBuilder creates an instance of Element, and sets the tag.
4. DOMCreator parses attributes, if any, and passes them to DOMBuilder.
5. DOMBuilder adds attributes to the Element.
6. DOMCreator parses text, if any, and passes it to DOMBuilder.
7. DOMBuilder adds text.

After these steps, the client can fetch the element by calling getElement on DOMCreator.

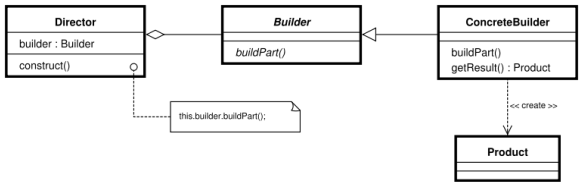

Figure 2.9: Structure of Builder

2.3.3 Composite

The Composite pattern "[c]ompose objects into tree structures to represent part-whole hierarchies." [**?**]. This a method of abstracting the types of a complex structure, and streamlining certain procedures. In listing 2.29 we've continued with the DOM, and created a structure that represents DOM elements that can be used to generate HTML.

In this example we have two Composites (DOMComposite, DOMElement) and one Leaf (DOMText). The client gets the html by calling the method getHtml on any of the elements desired, and they will take care of producing the result from all nested, if any, elements.

Listing 2.28: An example of implementation of Builder in JS

```
var DOMElement = {
    attributes = {},
    tag = null,
    text = ""
  },
  DOMBuilder = function (tag) {
    this.element = Object.create(DOMElement);
    this.element.tag = tag;
    this.addAttribute = function (key, value) {
      this.element.attributes[key] = value;
    };
    this.addText = function (text) { this.element.text = text; };
  },
  tokens = {}, // a map of tokens to parse
  fetch = function (str, token) {}, // returns specified type of token
  remove = function (str, token) {}, // removes token, returns modified
     string
  test = function (str, token) {}, // tests for specific token, return
     boolean
  DOMCreator = function (str) {
    var key, tag, text, value;
    // fetches the tag
    this.builder = new DOMBuilder(tag);
    while (test(str, tokens.whitespace)) {
      // fetches key-value pair of attributes, if any
      this.builder.addAttribute(key, value);
    }
    if (!test(str, tokens.slash)) {
      // fetches text, if any
      this.builder.addText(text);
    }
    // We have what we need
  };
DOMCreator.prototype.getElement = function () {
  return this.builder.element;
}
var element = new DOMCreator("<p>42</p>");
console.log(element.getElement()); // logs { attributes: {}, tag: "p", text
    : "42" }
```

2.3.4 Decorator

The Decorator pattern "[a]ttach additional responsibilities to an object dynamically" [**?**]. As JS is dynamic in its nature, this isn't a very difficult pattern to implement. In listing 2.30, we've simplified the example used by Addy Osmani in his book Learning JavaScript Design Patterns[31].

To use the participants in figure 2.11, we have PC as the ConcreteComponent, and addMemory, addScreen, and addKeyboard as the ConcreteDecorators.

[31] http://addyosmani.com/resources/essentialjsdesignpatterns/book/#decoratorpatternjavascript

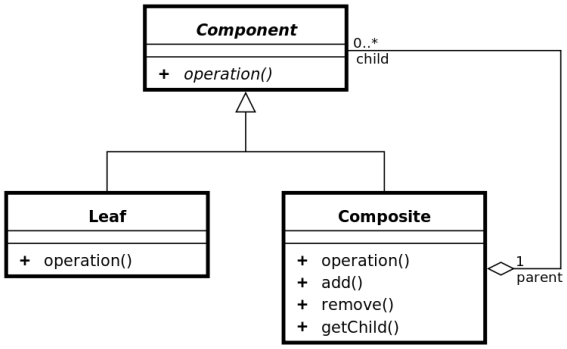

Figure 2.10: Structure of Composite

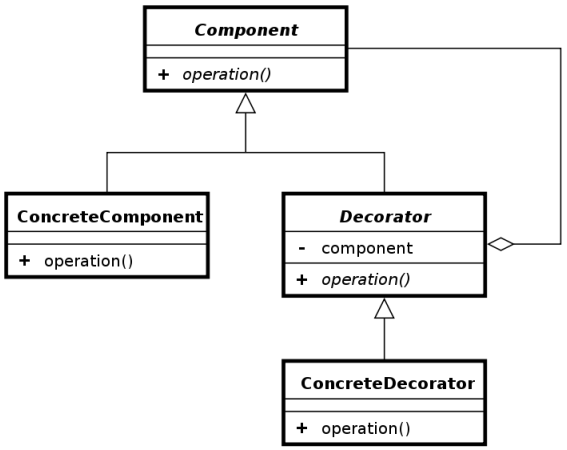

Figure 2.11: Structure of Decorator

2.3.5 Facade

The Facade pattern "[p]rovide a unified interface to a set of interfaces in a subsystem" [**?**]. Again, jQuery shows us an example of design pattern, as the constructor of the jQuery-object applies the Facade pattern. It's usually used to simplify the API the user have to concern himself/herself with, by delivering a subset of methods from underlying modules.

In listing 2.31, we've designed an object that takes the libraries jQuery and when.js, and delivers a new interface that taps into some of their functionality. The facade in the example have one method, namely load, and promises to load the callback functions in the order they're used (i.e. `http://example.org/1337` will not be loaded before `http://example.org/42` has completed).

2.3.6 Proxy

The Proxy pattern "[p]rovide a surrogate or placeholder for another object to control access to it" [**?**]. There are several kinds of proxies, like the virtual proxy (works like a lazy instantiator,

Listing 2.29: An example of implementation of Composite in JS

```
var DOMComposite = function (children) {
    this.children = children;
  },
  DOMElement = function (tag, content) {
    this.tag = tag;
    this.content = content;
  },
  DOMText = function (text) {
    this.text = text;
    getHtml: function () {
      return this.text;
    }
  };
DOMComposite.prototype = {
  addChild: function (element) {
    this.children.push(element);
  },
  getHtml: function () {
    var child, html = "";
    for (child in this.children) {
      html += child.getHTML();
    }
    return html;
  }
}
DOMElement.prototype.getHtml = function () {
  var text = "<" + this.tag;
  if (this.content) {
    return text + ">" + this.content.getHtml() + "</" + this.tag + ">";
  }
  return text + "␣/>";
};
DOMText.prototype.getHtml = function () {
  return this.text;
}
var text1 = new DOMText("42"),
    text2 = new DOMText("1337"),
    composite1 = new DOMComposite([ text1 ]),
    element1 = new DOMElement("span", text2),
    composite2 = new DOMComposite([ composite1, element1 ]);
console.log(composite2.getHtml()); // logs "42<span>1337</span>"
```

i.e. only creating the proxied object when you need it), remote proxies (proxies an object on a remote destination), and controlling proxies (to handle access), and they may be combined.

In listing 2.32 we've constructed a remote proxy. The object it proxies has one single purpose, which is to fetch the resource located at a given IRI. This may be one way of circumventing SOP.

Listing 2.30: An example of implementation of Decorator in JS

```
var PC = { cost: function () { return 1000; } },
    addMemory = function (PC) {
      return PC.cost: function () { PC.cost() + 300; };
    },
    addScreen = function (PC) {
      return PC.cost: function () { PC.cost() + 30; };
    },
    addKeyboard = function (PC) {
      return PC.cost: function () { PC.cost() + 7; };
    },
    myPC = Object.create(PC);
addMemory(myPC);
addScreen(myPC);
addKeyboard(myPC);
console.log(myPC.cost()); // logs 1337
```

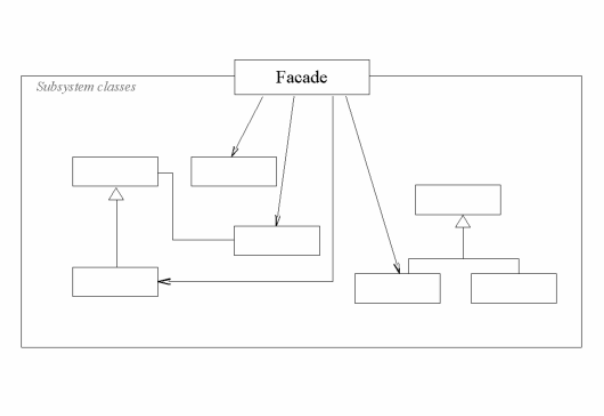

Figure 2.12: Structure of Facade

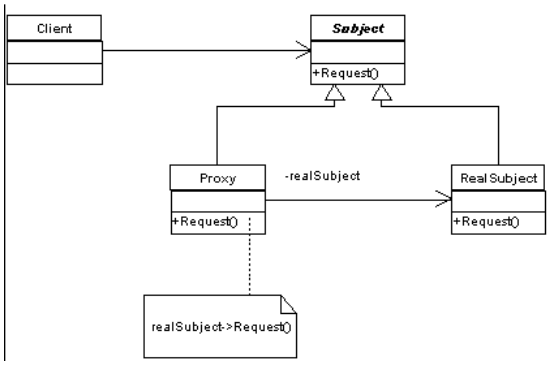

Figure 2.13: Structure of Proxy

2.3.7 Singleton

The Singleton pattern "[e]nsure a class only has one instance, and provide a global point of access to it" [**?**]. This might be helpful if we know there's no need for multiple instances of an object, or we need to handle all handling of data through one, central instance.

Listing 2.31: An example of implementation of Facade in JS

```
// assumes $ and When are global variables
var facade = (function (jQuery, When) {
  var promise = null;
  function load (uri, callback) {
    promise = When.defer();
    jQuery.get(uri, {}, function () {
      promise.resolve(arguments);
      callback.apply(this, arguments);
    });
  }
  return {
    load: function (uri, callback) {
      if (promise) {
        promise.then(function () {
          load(uri, callback);
        });
      } else {
        load(uri, callback);
      }
    }
  };
}($, When));
facade.load("http://example.org/42", function () {
  console.log(42);
});
facade.load("http://example.org/1337", function () {
  console.log(1337);
});
// Console will always log 42 first, 1337 second
```

Listing 2.32: An example of implementation of Proxy in JS

```
// assumes function ajax, that acts like $.ajax
var proxy = function (iri, callback) {
  ajax(/localproxy/, {
    data: {
      iri: iri
    },
    method: get,
    success: callback
  });
};

proxy("http://another.domain.com/42", function (data) {
  console.log(data); // logs whatever was fetched from http://another.
     domain.com/42
});
```

We've implemented a singleton in listing 2.33 that allows us to set and retrieve value for an object that's shared between multiple variables.

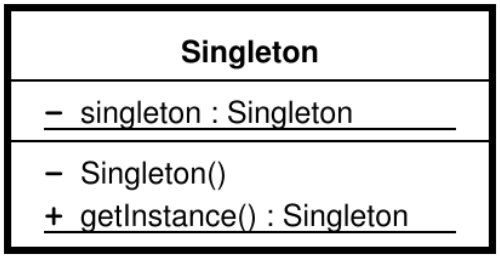

Figure 2.14: Structure of Singleton

Listing 2.33: An example of implementation of Singleton in JS

```
var Singleton = (function () {
      var instance;
      return function () {
        instance = instance || Object.create({
                  getValue: function () { return this.val; },
                  setValue: function (val) { this.val = val; }
                });
        return instance;
      };
    }()),
    objA = Singleton,
    objB = Singleton;
objA.setValue(42);
console.log(objB.getValue()); // logs 42
```

2.3.8 Strategy

The Strategy pattern "[d]efine a family of algorithms, encapsulate each one, and make them interchangeable" [**?**]. It relies on a shared interface of properties among objects, and an example is shown in listing 2.34[32].

This pattern streamlines our functionality by eliminating conditional statements.

2.4 Test-driven development (TDD)

TDD is a development process of software that details how to build your codebase. It relies on a iterative cycle of steps that are summarized in figure 2.16. The process start with writing a test that asserts the functionality we wish to implement. It should raise a red flag when first tested (meaning that the functionality isn't implemented yet), which in turn leads us to implement the requested feature. When we manage to get a green flag (meaning that the functionality now

[32]Inspired by the example given by Mike Pennisi in his blogpost at `http://weblog.bocoup.com/the-strategy-pattern-in-javascript/`

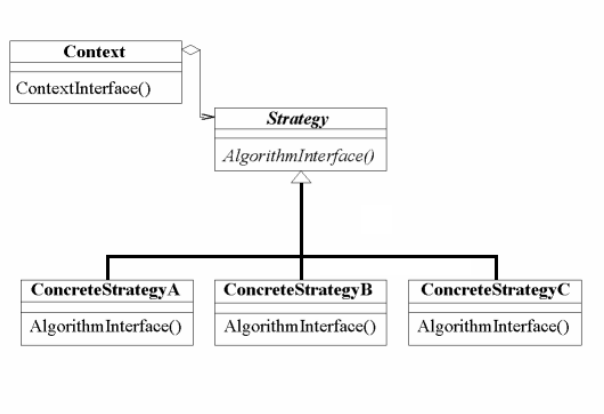

Figure 2.15: Structure of Strategy

Listing 2.34: An example of implementation of Strategy in JS

```
var buttons = [{
      id: "Button42",
      onclick: function () {
        console.log(42);
        return false;
      }
    }, {
      id: "Button1337",
      onclick: function () {
        console.log(1337);
        return false;
      }
    }],
    button;
for (button in buttons) {
  document.getElementById(button.id).onclick(button.onclick);
}
```

exists), we can continue to either write a new test for a new functionality, or we can refactor the excisting code by making sure it doesn't raise any red flags (break any tests).

TDD are discussed in section 6.7, to discuss things we've learned in the progress of implementing the framework.

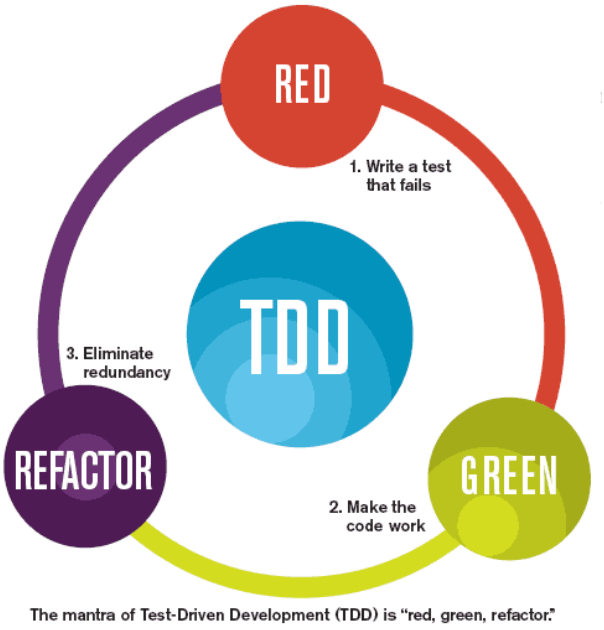

Figure 2.16: An illustration of the TDD-process.

Chapter 3

Problem Description and Requirements

The notion of RDF as a standard for exchanging structured data on WWW is becomming increasingly popular. The technologies of SW are actively developed, and new features, as well as stabilizing old ones, are in the works.

JS has also gotten a lot of attention as an increasingly powerful programming language for WWW. First and foremost as a clientside scripting language, but now also as serverside implementations. Large companies like Google, Microsoft, Mozilla, Apple, and Opera are all putting a lot of effort into increasing the effectiveness of their JS-engines, through implementations and cooperating in evolving the standards.

In this environment, you would think that many developers would try to access SW with a library written in and for JS. While there are projects trying to create frameworks for accessing, query, and manipulating SW, none of them are the defining prototype of a JS-framework for SW as of yet.

3.1 Problem

This thesis seeks to define what is needed in order to have a powerful framework in JS that can access SW. This goal is divided into four subgoals:

1. The first subgoal is to identify the features that a framework accessing SW needs to support. Which technologies needs to be involved, what obstacles do they introduce, and what are the consequences of implementing them?

2. Next, we need to identify the participants and how they should collaborate. We'll try using the knowledge of SDP to describe the components in known and widely utilized language. While doing this, we need to explore how JS conform to the patterns of the various SDPs.

3. Our third subgoal is to implement a functional framework in JS. In doing this, we need to identify what features JS offers us that are relevant for our framework.

4. Our final goal is to develop APIs that exposes the functionality of the framework in a way that is easy for developers to get into.

3.2 What are the components required for the framework?

We've identified a lot of the technologies regarding SW in section 2.1. RDF and its serializations need to be a part of the framework. As documents containing RDFS, OWL, and other vocabularies, are subsets of RDF, a representation of RDF should be enough in terms of representing the model.

After we've implemented a model of RDF, we need to implement some way of letting developers browse, or query, our data. SPARQL is a powerful language that handles this purpose, but does it raise the bar for using the framework unnecessarily high? Will developers new to SW want to tackle SPARQL in addition to all the other technologies they need to learn?

Another important feature of the framework will be the APIs. How should we expose the functionality to developers? Should it be available as one monolithic object, or is there a need to divert it into several smaller objects? This problem is further investigated in section 3.5.

3.3 Which Design Patterns are applicable for the components?

We've decided to use SDP to help us decide how we should model our components. This will hopefully help us in identifying participants, collaborations between the participants, and the consequences of implementing them.

Section 2.3 explained in detail the SDPs we think are appropriate for the framework. But these patterns were originally developed for class-oriented programming languages (e.g. C++, which was used to write the sample code used in Design Patterns). Is it appropriate to apply these patterns to a class-less programming language like JS? And if not fully compatible, are there ways to "tweak" the premises, so that we may use the amassed knowledge of patterns to our advantage?

3.4 Which JS-functionalities are of use for the framework?

What are the challenges the framework will have to deal with when implementing the required components? And how does JS align with these problems? We've described JS in section 2.2 and featured some of the functionality that are relevant for our framework.

Serializations will most likely need to be loaded asynchronously. This is handled by the asynchronous loading capacities described in section 2.2.5. We also need certain functions to be called in correct order in response to the asynchronous functionality. By making use of the functional features of JS (section 2.2.1.3), in combination with the promise pattern (section 2.2.6.2), we believe this to be achievable.

3.5 How should the API be designed?

Although SDPs give us a lot of hints as how to design our components, most of the signature of the objects are still up to grab. What are the possibilities we have within the restrains of SW and JS. What is the most effective API to expose to JS-developers that wish to harness the structural data in SW?

How big should it be? How much of the API should be public, i.e. how granular should our functions be? Should it be modularized, or just offer one monolithic API?

These are questions we hope to offer an answer to in our implementation of an actual, working framework.

Chapter 4

Tools

This chapter will describe the 3rd party libraries, software, and services we've used in this thesis.

4.1 rdfstore-js (RDFStore)

RDFStore describes itself as "a pure Javascript implementation of a RDF graph store with support for the SPARQL query and data manipulation language"[1]. It supports a range of features:

- Works in browsers as well as serverside
- Full support of SPARQL 1.0 and SPARQL Update Language, and partially SPARQL 1.1 (including partial support for property paths).
- Both JSON-LD, Turtle, and N3 parsers.
- Implemented W3C RDF Interfaces API[2].
- Have an experimental implementation of RDF graph evens API.
- Custom filter functions.
- Threaded API if WebWorkers are supported.
- Persistent storage with HTML5 LocalStorage (for browsers) or MongoDB (for Node.js (Node)).
- Implementation of the SPARQL Protocol for RDF[3] on the serverside.

[1] https://github.com/antoniogarrote/rdfstore-js/#readme
[2] http://www.w3.org/TR/rdf-interfaces/
[3] `http://www.w3.org/TR/rdf-sparql-protocol/`

Graphite has used some of the components of this library (12 out of 35), and rewritten them to fit into the overall system. The components that have been used can be seen in table 5.1.

RDFStore is available as a repository at GitHub (GH)[4], and has a nice pace of development. It's author is Antonio Garrote, and it also lists Christian Langanke as contributor.

4.2 rdfQuery (RDFQuery)

RDFQuery describes itself as "an easy-to-use Javascript library for RDF-related processing". It depends on jQuery, and is distributed in three versions:

1. Core rdfQuery: Creates and queries triplestores.
2. rdfQuery with RDFa: Parses RDFa.
3. rdfQuery with rules: Enables reasoning with rules.

Graphite has integrated the its parsers as well as several utility-components. A complete list can be seen in table 5.1.

RDFQuery has several contributors to its official project-page[5], which is led by Jeni Tennison, Rene Kapusta, and Haymo Meran. After a long time of development seemingly going dead, it now has a repository on GH[6], which seems to be led by Sebastian Germesin. The repository is a mirror of the original project, and all contribution to the GH-project are contributed back to the official project-page.

4.3 Buster.JS (Buster)

Buster is a "JavaScript test framework for node and browsers". It has been in beta for a couple of years, and shows promising results. They hope to have it ready by the end of this summer.

Some important features of Buster are:

- Supports tests for Node and browsers: You can use the same testcases for both environments, but also create dependencies by the use of feature detection.
- Assertions, refutations, stubs and spies, expectations, events, properties, and supporting utilities and objects; There are a lot of possible ways to test your code.

[4] `https://github.com/antoniogarrote/rdfstore-js/`
[5] `http://code.google.com/p/rdfquery/`
[6] `https://github.com/alohaeditor/rdfQuery`

- Flexibility: There's a lot of public APIs, which can be used to write specific code as needed.
- Extensibility: There are already several extensions available, such as buster-amd and buster-lint.
- Asynchronous testing: Buster can test resources that only are available in asynchronous functions.
- Structuring: Nested test cases, setup, and teardown helps you structure and reuse code easier.
- Deferred tests: Easy seclusion of tests that clutters your reports, e.g. while refactoring.
- Measuring time: It reports in milliseconds how long each testrun took.

Graphite has used Buster extensible throughout its development, and now sports 36 test cases, with 566 tests, with a total of 1990 assertions. It's all being run in about 10 seconds.

Buster is available at GH[7], and is being led by August Lilleaas and Christian Johansen.

4.3.1 Node.js (Node)

Buster is dependendent on Node, and as such has used it throughout the development. Node is "a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications.

Node supports a great list of features, amongst them:

- A simple module loading system
- Networking utilities: Setting up servers (e.g. with HTTP, Hypertext Transfer Protocol Secure (HTTPS) and net), creating sockets (e.g. with net and User Datagram Protocol (UDP)), lookup Domain Name System (DNS), handling URLs and queries.
- Filesystem utilities: File I/O are simplified with modules such as File System, path, and os.
- Binary Data utilities: APIs for handlings streams and buffers, easing the handling of transmitting binary data on WWW.
- Much, much more.

Node is available at `http://nodejs.org` and latest versions at GH[8]. Originally created by Ryan Dahl in 2009, it's now sponsored by Joyent, his employer, and enjoys contributions of many, many developers.

[7] `https://github.com/busterjs/buster`
[8] `https://github.com/joyent/node`

4.4 RequireJS (Require)

Require is ”a JavaScript file and module loader“. It allows modularization of JS by deploying functionality throughout several files, or modules. It ties it all together by supporting technologies such as AMD (explained in section 2.2.6.1).

Below are some of the features of Require:

- Loading of modules through AMD for browsers, Rhino (an implementation of JS written in Java[9]), and Node.
- Code optimizing
- Threading by utilizing Web Workers

Graphite has used Require to include the vast list of modules as they are needed.

4.5 when.js (When)

When is a ”lightweight CommonJS Promises/A and when() implementation“. It’s allows usage of the Promises pattern (section 2.2.6.2), and also provides several other useful Promise-related concepts. It’s being developed by Brian Cavalier, and is available at GH[10].

The module named Promise is an integration of When into Graphite.

4.6 Git

Git is a free and open source Distributed Version Control System (DVCS), originally created by Linus Torvalds as a response to existing Version Control Systems (VCSs) and Source Control Managements (SCMs). It’s fast, relient, and relatively easy to use. It offers a variety of features, such as:

- Setup, configuring, getting and creating projects.
- Branching and merging, sharing and updating.
- Inspect and compare code.
- Much, much more[11].

Graphite has used Git to share its codebase, making it available for all to use and contribute to.

[9] `http://www.mozilla.org/rhino/`
[10] `https://github.com/cujojs/when`
[11] For complete list, see the documentation at `http://git-scm.com/docs`

4.6.1 GitHub (GH)

GH is a service that lets you share and collaborate with developers[12], be they friends or strangers. It's a social network that uses Git as its technological foundation, making it easier to connect with other developers. It supports features such as:

- User and organization-accounts.
- Free git-repositories (as long as they are open source).
- Secure transmissions.
- Utilities for presentation of text and code, such as pages, wikis, gists (usually to present snippets of code), issues management, and graphs (visual presentations of data, such as contributors, commit activity, etc).

Graphite has used GH as a central repository for its codebase, which is available at `https://github.com/megoth/graphitejs`.

4.7 WebStorm (WS)

WS is a JS Integrated Development Environment (IDE) that is available for Windows, Mac OS, and Linux. It offers a wide array of tools for developing with JS, such as:

- Refactoring.
- Structuring code consistently.
- Integration with GH, Node, and JsTestDriver (test framework for JS, developed by Google).
- Smart duplicated code detector.
- Much, much more[13].

The development of Graphite has been done primarily in WS.

WS is being developed by JetBrains, is primarily licensed, but offers free licenses for educational and open source purposes. It's available at `http://www.jetbrains.com/webstorm/`.

[12] `https://github.com/about/`
[13] http://www.jetbrains.com/webstorm/features/index.html

Chapter 5

The Graphite Framework

This chapter will list all the modules that have been implemented. They are listed alphabetically, based on their names. The names shoould reflect their purpose, and as such should give an intuitive hint of what they can do.

Apart from their names, all modules have a list of features that are presented in the beginning of their section. The list contains the following attributes:

1. Branch: The branch in which they reside (explained below)
2. Location: The adress to which they are located in the src-folder
3. Dependencies: Lists which modules, if any, that the module are dependent on
4. Size: The size in kilobytes (kBs)
5. Source lines of code (SLOC): The number of source code lines
6. Design Pattern: The design pattern(s) that have been used as a starting point for this module, if any (not applicable for 3rd-party derived modules)
7. Test result: If tests are written for the module, its results are listed here

All modules are divided into one of three branches; Graphite, RDFQuery, or RDFStore (as shown in table 5.1). The two latter are originally taken from 3rd-party libraries (explained in Tools, Chapter 4), and modules residing in these will note in which degree they have been modified.

After the initial block detailing the attributes, a lengthier description explains the module in whole. Considerations taken will be noted, and variations/possibilities are explained.

Dependencies between the main modules have been visualized in figure 5.1. The dependencies within each subdomain of modules are listed near their appropriate main module.

Graphite	RDFQuery	RDFStore
API (page 55)	CURIE (page 56)	Abstract Query Tree (page 57)
Graph (page 60)	Datatype (page 56)	B-Tree (page 55)
Graphite (page 62)	RDF/XML (page 70)	Backend (page 61)
JSON-LD (page 69)	Turtle (page 70)	Callbacks (page 58)
Loader (page 63)	URI (page 71)	Engine (page 57)
Proxy (page 64)		Lexicon (page 62)
Promise (page 71)		Query Filters (page 59)
Query (page 64)		Query Plan (page 59)
Query Parser (page 65)		RDF JS Interface (page 60)
RDF (page 67)		Tree Utils (page 71)
RDF JSON (page 69)		
RDF Loader (page 68)		
RDF Parser (page 68)		
SPARQL (page 66)		
SPARQL Full (page 66)		
Utils (page 72)		
XHR (page 64)		

Table 5.1: Overview of which branches each module belongs to

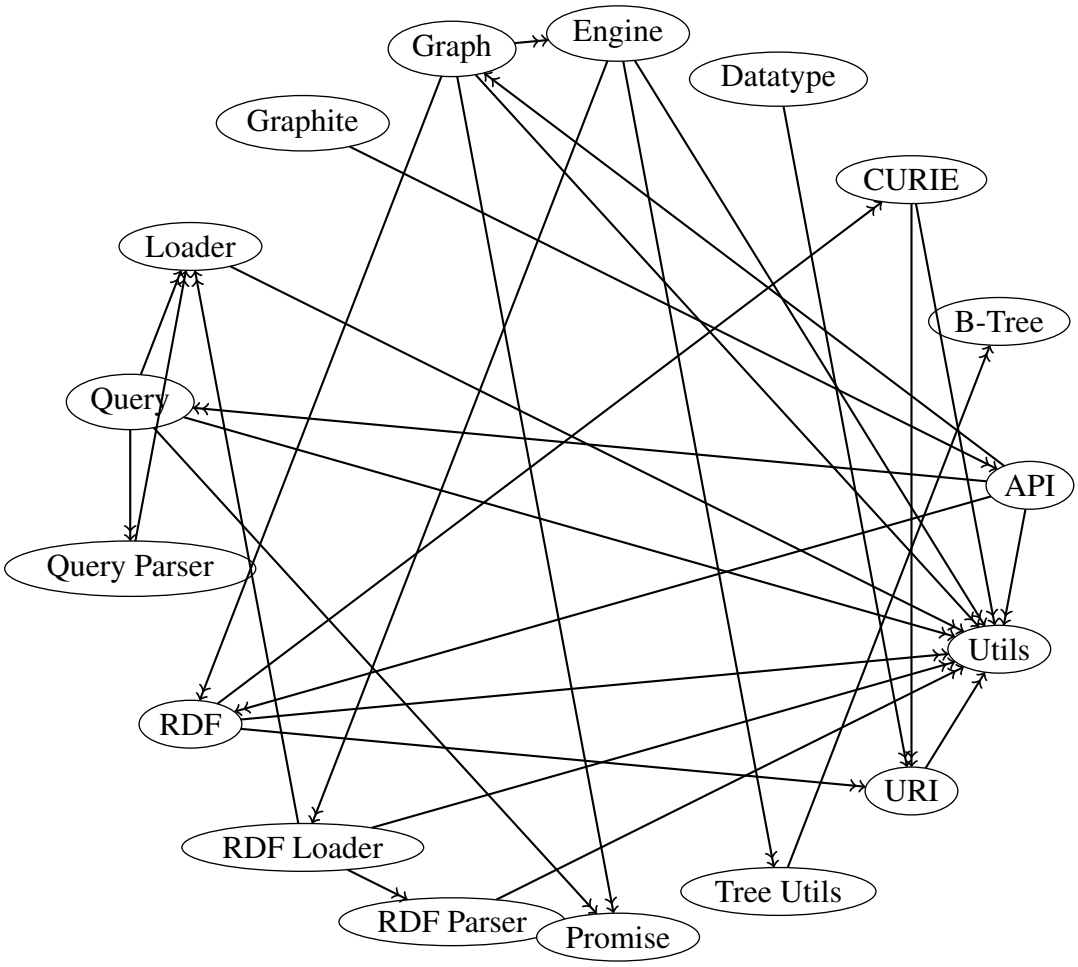

Figure 5.1: Dependencies between the main modules of Graphite.

The source can be forked at `https://github.com/megoth/graphitejs`, or read in appendix 7.1 (page 79).

5.1 API

Branch	Graphite
Location	`graphite/api.js`
Dependencies	Graph (page 60) , Query (page 64) , RDF (page 67) , Utils (page 72) , Promise (page 71)
Size:	4.4 kB
SLOC	121
Design Pattern	Bridge (page 31) , Facade (page 35)
Test result	10 tests, 11 assertions; Total average: 1968, Average/assertion: 179

The API module tries to combine the most powerful modules of Graphite into one module, for easier access to developers new to the framework. The idea is to lower the barrier by combining several modules into one, and built upon their functionality to create new ways of handling the data.

This module differs from the Graphite module in that it acts as a Facade-object for the underlying modules, instead of a simple connection to them. Its signature mirrors in many ways it underlying modules, but adds some methods of it's own. In most of the cases though the mapping are one-to-one, which should make a transition from the API module to the Graph- or Query module easy.

At the heart of the module is the properties g and q, which respectively are instantiations of the Graph- and Query module. This core properties enables the user to cache data from the SW, and query it. The query can be built piece by piece, until it are executed with the execute-method.

This module partakes in the Bridge pattern as the Implementor, where Graphite (page 62) works as the Abstraction. It is designed to be easily switched if a system architect wishes to customize the API he wants to serve his team of developers.

5.2 B-Tree

Branch	RDFStore
Location	`rdfstore/rdf-persistence/in_memory_b_tree.js`
Dependencies	None
Size:	27.8 kB
SLOC	547
Design Pattern	None
Test result	4 tests, 152 assertions; Total average: 55, Average/assertion: 0

The B-Tree module is an implementation of a generic B-tree, more specifically an adaptation of one made for C[1]. It doesn't make use of any SDPs.

5.3 CURIE

Branch	RDFQuery
Location	`rdfquery/curie.js`
Dependencies	URI (page 71), Utils (page 72)
Size:	10.5 kB
SLOC	84
Design Pattern	None
Test result	13 tests, 13 assertions; Total average: 61, Average/assertion: 5

The CURIE module handles functions regardings Compact URIs (CURIEs)[2], which are quite common when working with SW, as it eases the task of remembering IRIs, in turn helping to reduce typing errors. The functions are split into creating IRIs from CURIEs or vice versa.

CURIE isn't written with a SDP in mind, as it's taken from a 3rd party library. It could be argued it's a utilization the Builder pattern, but the strings it returns can hardly be called complex objects.

5.4 Datatype

Branch	RDFQuery
Location	`rdfquery/datatype.js`
Dependencies	URI (page 71)
Size:	14.5 kB
SLOC	260
Design Pattern	Strategy (page 39)
Test result	12 tests, 22 assertions; Total average: 56, Average/assertion: 3

The Datatype module returns a simple function that returns an object representing a datatype. It's used by the RDF module when handling literals. In addition to the callable function there's also a static function *valid* available, that enables us to test whether or not a given value is valid according to a given datatype.

The module uses the Strategy pattern within itself (i.e. no collaboration with other modules). This is done by giving the different datatypes each a representation with an object containing the properties *regex*, *strip*, and *value*, and in some cases *validate*. The Context is in this case

[1] `http://www.gossamer-threads.com/lists/linux/kernel/667935`
[2] `http://www.w3.org/TR/curie/`

either the constructor-function, or the static function valid.

5.5 Engine

Branch	RDFStore
Location	`rdfstore/query-engine/query_engine.js`
Dependencies	Abstract Query Tree (page 57) , Tree Utils (page 71) , Query Plan (page 59) , Query Filters (page 59) , RDF JS Interface (page 60) , RDF Loader (page 68) , Callbacks (page 58) , Utils (page 72)
Size:	71.7 kB
SLOC	1543
Design Pattern	Builder (page 32) , Facade (page 35)
Test result	53 tests, 312 assertions; Total average: 718, Average/assertion: 2

The Engine module is a complex module that brings together several submodules, in effect being an implemention of the Facade pattern. Its purpose is to execute queries, and does so by iterations of compiling data, that results in either a formula (as specified in RDF (page 67)), a list of objects with projected variables, or a boolean (depending on the query form, as explained in section 2.1.6.1).

The Builder pattern can be used to understand this module, although it's somewhat hazy. The engine participates as the Director, and RDF JS Interface, Query Filter, and Query Plan all collaborate as Builders. The Product in this case is the result of a query, and it is here it becomes clear that the implementation is not complete, as it is the engine itself that serves the means of getting the Product.

Figure 5.2 shows the dependencies amongst the submodules of the Engine module. Some of the main modules are also represented (i.e. Query Parser, RDF loader, Tree Utils, and Utils), as they've been used by the submodules.

5.5.1 Abstract Query Tree

Branch	RDFStore
Location	`rdfstore/query-engine/abstract_query_tree.js`
Dependencies	Query Parser (page 65) , Tree Utils (page 71) , Utils (page 72)
Size:	27.1 kB
SLOC	540
Design Pattern	None
Test result	18 tests, 90 assertions; Total average: 130, Average/assertion: 1

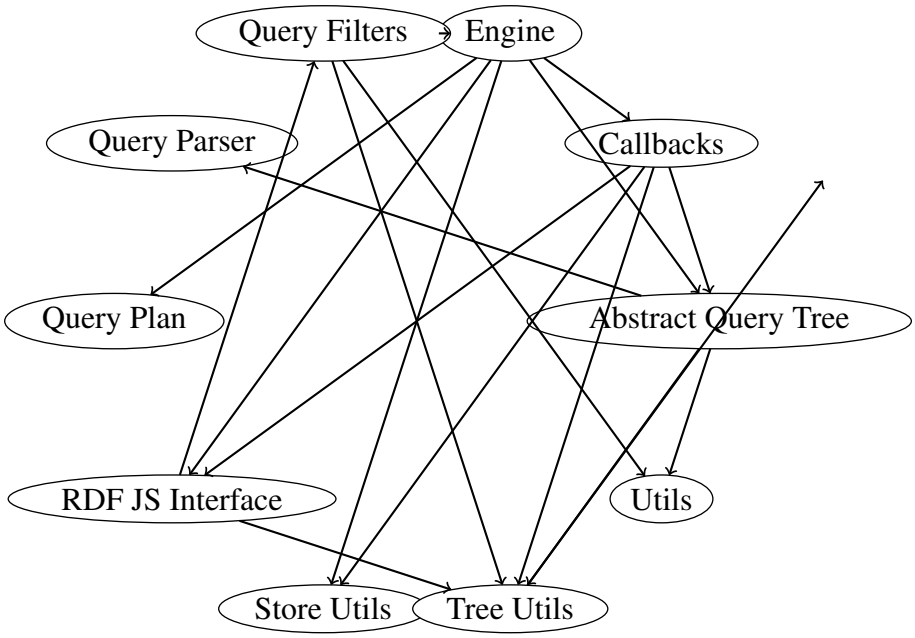

Figure 5.2: Dependencies between the Engine-related modules of Graphite.

The Abstract Query Tree module is based on the draft of The SPARQL Algebra[3], and doesn't apply any SDPs as we can see. Again, the code align closely to the Builder pattern, and it shouldn't be to hard to refactor the module. Another pattern that could easily be applied is the Strategy pattern. We'll return to this point in the discussion (section 6.1.1).

5.5.2 Callbacks

Branch	RDFStore
Location	`rdfstore/query-engine/callbacks.js`
Dependencies	Abstract Query Tree (page 57) , RDF JS Interface (page 60) , Tree Utils (page 71)
Size:	18.8 kB
SLOC	437
Design Pattern	Builder (page 32)
Test result	7 tests, 25 assertions; Total average: 102, Average/assertion: 4

The Callbacks module is a submodule of the Engine module, and handles the order in which queries should be fired. The module participates in the Builder pattern in collaboration with RDF JS Interface (page 60) , where Callbacks works as the Director, and Interface works as the Builder. It does it with a twist though, as explained in section 5.5.2.

It could also make use of the Observer pattern, which is discussed in section 6.8.2.

[3] `http://www.w3.org/2001/sw/DataAccess/rq23/rq24-algebra.html`

5.5.3 Query Filters

Branch	RDFStore
Location	`rdfstore/query-engine/query_filters.js`
Dependencies	Tree Utils (page 71) , Utils (page 72)
Size:	79.3 kB
SLOC	1435
Design Pattern	Builder (page 32)
Test result	15 tests, 29 assertions; Total average: 186, Average/assertion: 6

The Query Filters are utility functions for handling queries. It handles aggregation, function calls, and other filter expressions as part of a SPARQL abstract tree. As such, the module acts as a Builder for the engine, which acts as a Director, meaning that the module partakes in the Builder pattern.

The module could also benefit from the use of the Strategy pattern, as many of the filter expressions could be handled as interchangeable objects.

5.5.4 Query Plan

Branch	RDFStore
Location	`rdfstore/query-engine/query_plan_sync_dpsize.js`
Dependencies	None
Size:	21.6 kB
SLOC	468
Design Pattern	Builder (page 32)
Test result	1 tests, 12 assertions; Total average: 9, Average/assertion: 1

The Query Plan module handles the different ways you can consolidate the different parts of a SPARQL abstract tree. It takes part of the Builder pattern in collaboration with the Engine, Query Filter, and RDF JS Interface, as previously explained.

An adoptation of the Strategy pattern would probably clean up the structure, as well as the Composite pattern.

5.5.5 RDF JS Interface

Branch	RDFStore
Location	`rdfstore/query-engine/rdf_js_interface.js`
Dependencies	None
Size:	20.2 kB
SLOC	536
Design Pattern	Builder (page 32)
Test result	5 tests, 20 assertions; Total average: 79, Average/assertion: 4

The RDF JS Interface module implements the API defined in the document RDF Interfaces[4]. It outlines many common RDF terms and sports some functions to help creating them. In this we see what resembles an implementation of the Builder pattern (as mentioned in the Engine and Callbacks module).

5.6 Graph

Branch	Graphite
Location	`graphite/graph.js`
Dependencies	Backend (page 61), Engine (page 57), Lexicon (page 62), Promise (page 71), RDF (page 67), Utils (page 72)
Size:	11.7 kB
SLOC	261
Design Pattern	Strategy (page 39)
Test result	4 tests, 8 assertions; Total average: 1019, Average/assertion: 127

The Graph module is the cornerstone of Graphite. It is the abstraction of quadstores, and serves as an accesspoint for all the data processed by the framework. It's signature is somewhat small, but what it powers is the execution of SPARQL-queries. By calling it's method execute with a query (be it an instantiation of the module Query, or a plain String) you can add and retrieve data.

To limit the scope of the thesis, we decided to only support a subset of the SPARQL Query Language and SPARQL Update. The subset are:

- Query Forms
  - ASK
  - CONSTRUCT
  - INSERT

[4] `http://www.w3.org/TR/rdf-interfaces/`

  - LOAD
  - SELECT
- Solution Sequences and Modifiers
  - ORDER BY
- Aggregates
  - GROUP BY
- Aggregate Algebra
  - Count
  - Sum
  - Avg
  - Min
  - Max

This means we can only add data to the graph, not delete, clear, or update it. Also, subqueries are not supported. This limitation was made to avoid some common problems when dealing with logics in quadstores, as well as limitation imposed by underlying 3rd-party code.

The Strategy pattern has been implemented in order to handle the supported forms of queries. It's handled internally, with the function execute fetching a concrete strategy from a map of functions (e.g. *executes["select"]* contains the function that handles results from SELECT queries).

An important feature of the Graph module is lazy loading, which secures that the order in which we call resources are handled correctly. This works by making use of the Functional Feature (section 2.2.1.3) combined with the Promise Pattern (section 2.2.6.2). Lazy loading as a design pattern is discussed in section 6.8.1.

The Graphs' and submodules' dependencies are listed in figure 5.3 (Tree Utils are present to show external dependencies).

5.6.1 Backend

Branch	RDFStore
Location	`rdfstore/persistence/quad_backend.js`
Dependencies	Tree Utils (page 71) , Tree Utils (page 71)
Size:	4.0 kB
SLOC	89
Design Pattern	None
Test result	2 tests, 57 assertions; Total average: 13, Average/assertion: 0

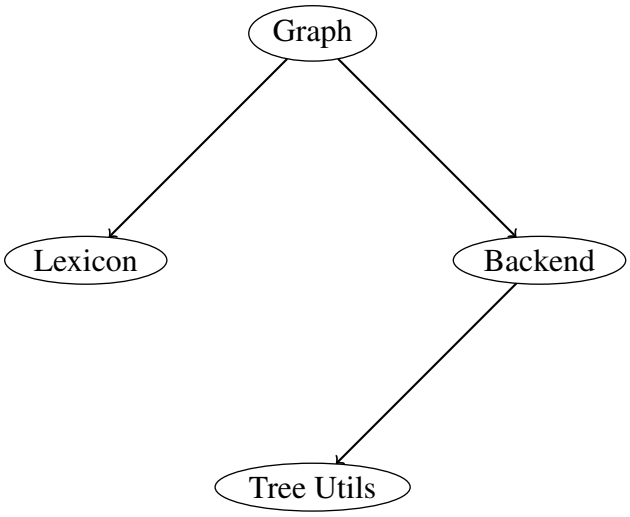

Figure 5.3: Dependencies between the Graph-related modules of Graphite.

The Backend module handles the storage of RDF-related data in one graph. It makes use of no SDPs.

5.6.2 Lexicon

Branch	RDFStore
Location	`rdfstore/persistence/lexicon.js`
Dependencies	None
Size:	9.8 kB
SLOC	242
Design Pattern	None
Test result	2 tests, 9 assertions; Total average: 11, Average/assertion: 1

The Lexicon module handles all the graphs that are in play, and resolves terms across graphs. It makes no use of SDPs.

5.7 Graphite

Branch	Graphite
Location	`graphite.js`
Dependencies	API (page 55)
Size:	0.5 kB
SLOC	7
Design Pattern	Bridge (page 31)
Test result	1 tests, 2 assertions; Total average: 7, Average/assertion: 4

The Graphite module is designed to be the main entry point for beginners. It sits at the forefront

of the framework (all other modules resides in the folders that are its siblings), and is designed to be easily included into a larger context with an AMD-library. Developers can include this module without knowing anything about it, and start going through tutorials, the documentation, or just play around.

For now it merely returns the API module, but it can be easily extended. One way of doing this is to include the Utils module, which gives the extend-method for objects. By instantiating the different modules whose interface you wish to make highlight, you can combine them into one single object. This could be useful to a system architect who wishes to modify the framework for his project, minimizing the amount of time his developers need to spend to learn the framework. With his alteration he could simply present them with a modified API, that's scissored to their use.

5.8 Loader

Branch	Graphite
Location	`graphite/loader.js`
Dependencies	Proxy (page 64) , Utils (page 72) , XHR (page 64)
Size:	1.3 kB
SLOC	26
Design Pattern	Strategy (page 39)
Test result	1 tests, 1 assertions; Total average: 16, Average/assertion: 16

The Loader module fetches resources, and does so depending on what functionality the system supports. All dependent modules prefixed Loader participates in the Strategy Pattern as a ConcreteStrategy.

The dependencies within the submodules of Loader is shown in figure 5.4.

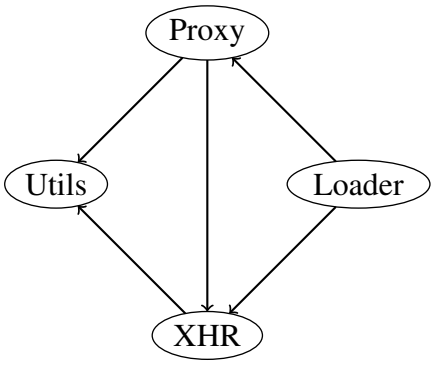

Figure 5.4: Dependencies between the Loader-related modules of Graphite.

5.8.1 Proxy

Branch	Graphite
Location	`graphite/loader/proxy.js`
Dependencies	XHR (page 64) , Utils (page 72)
Size:	1.2 kB
SLOC	36
Design Pattern	Bridge (page 31) , Proxy (page 35) , Strategy (page 39)
Test result	2 tests, 4 assertions; Total average: 26, Average/assertion: 7

The Proxy module is a participant in the Strategy Pattern as ConcreteStrategy. It was created to bypass the Same Origin Policy (section 2.2.5.1) by using a proxy-server on the same domain. It uses the XHR module to make this connection, and if successful, the service would return the data the framework would otherwise be denied.

This does require a service to be set up on the server, which accepts the formatted query which the Proxy sends. Basicly it split the IRI to be loaded into separate parts (as explained in figure 2.5). As part of the framework, this service has been created as an application driven by Node.

5.8.2 XHR

Branch	Graphite
Location	`graphite/loader/xhr.js`
Dependencies	Utils (page 72)
Size:	1.5 kB
SLOC	40
Design Pattern	Bridge (page 31) , Strategy (page 39)
Test result	4 tests, 10 assertions; Total average: 76, Average/assertion: 8

The XHR module makes use of the XHR2-object available in most modern browsers. It makes use of the Strategy Pattern by participating as a ConcreteStrategy to the Loader module.

5.9 Query

Branch	Graphite
Location	`graphite/query.js`
Dependencies	Loader (page 63) , Query Parser (page 65) , Utils (page 72) , Promise (page 71)
Size:	12.1 kB
SLOC	292
Design Pattern	Builder (page 32)
Test result	51 tests, 52 assertions; Total average: 512, Average/assertion: 10

The Query module builds a complex structure that aligns the SPARQL abstract tree, which is used in the Engine module. It partakes in the Builder pattern by being the Director-participant, whereas the Query Parser module (and its submodules) is the Builder-participant.

5.10 Query Parser

Branch	Graphite
Location	`graphite/queryparser.js`
Dependencies	SPARQL (page 66)
Size:	0.4 kB
SLOC	14
Design Pattern	Builder (page 32) , Strategy (page 39)
Test result	2 tests, 4 assertions; Total average: 10, Average/assertion: 2

The Query Parser module is designed to be extensible, i.e. if there are other ways of serialializing the abstraction of a SPARQL query, than it can be extended with this module (e.g. to differ between SPARQL 1.0 and SPARQL 1.1).

The module is designed with the Builder pattern in mind, by participating as the Builder. The Query module is Director, and decides in which order parts of the SPARQL abstract tree is to be added. It further delegates this responsibility to the chosen strategy, e.g. the module that participates as ConcreteStrategy (while the module itself participates as Context).

Figure 5.5 display the dependencies between the modules partaking in the works of the Query Parser.

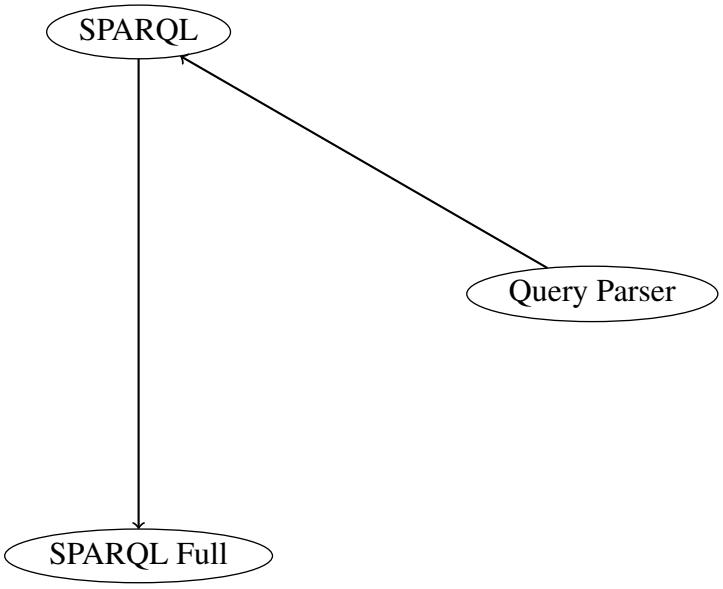

Figure 5.5: Dependencies between the modules related to the Query Parser in Graphite.

5.10.1 SPARQL

Branch	Graphite
Location	`graphite/queryparser/sparql.js`
Dependencies	SPARQL Full (page 66)
Size:	35.7 kB
SLOC	864
Design Pattern	Builder (page 32) , Strategy (page 39)
Test result	36 tests, 56 assertions; Total average: 124, Average/assertion: 2

The SPARQL module sports an array of methods that allows building parts of the SPARQL abstract tree. The tree has some constraints concerning how it can be structured, and the module takes care of this.

The module partakes in the Strategy pattern as a concrete strategy. As of now it is the only strategy available, which may make the use of the pattern unnecessary, and we'll return to this discussion in section 6.1.3.

The module also participates as a Builder in the Builder pattern. This responsibility is delegated from the Query Parser, which is also the same that acts as Context in the Strategy pattern.

As the module doesn't support parsing of all the elements in the SPARQL abstract tree, it also imports the use of SPARQL Full module.

5.10.1.1 SPARQL Full

Branch	RDFStore
Location	`rdfstore/sparql-parser/sparql_parser.js`
Dependencies	None
Size:	1000 kB
SLOC	19114
Design Pattern	Builder (page 32)
Test result	

The SPARQL Full module is by far the biggest component in Graphite. It's generated by PEG.js, which is a parser generator for JS[5]. That is also why it is much bigger and more complex than it needs to be. But it does parse a complete SPARQL query, and as we haven't been able to create a complete one ourselves, we've implemented it as part of our framework.

It works as a starting point for manipulating queries, by letting us feed it with a complete query, and then modify its parts as necessary through the Query module. As such, it's a part of the Builder pattern that's described with the other modules that takes part of the Query Parser.

[5] `http://pegjs.majda.cz/`

5.11 RDF

Branch	Graphite
Location	`graphite/rdf.js`
Dependencies	URI (page 71) . Utils (page 72)
Size:	16.8 kB
SLOC	372
Design Pattern	Composite (page 33) , Strategy (page 39)
Test result	7 tests, 17 assertions; Total average: 34, Average/assertion: 2

The RDF module offers a wide arsenal of methods, and creates a common ground for producing objects pertaining to terms in RDF. Many of the methods origins from the N3-parser in RDFStore, but has been restructured to promote a consistent API. As such, the method toNT is represented in all objects retrieved from RDF, and it presents the different terms in N3-compliant syntax (e.g. IRI = <(IRI)>).

The module is used by all the parsers, and the Engine is dependent on the toQuads-method it appends on all its objects. The objects available through RDF are:

- BlankNode
- Collection (e.g. a list)
- Empty (i.e. rdfs:nil)
- Formula (i.e. a set of statements)
- Literal
- Statement
- Symbol (e.g. a IRI)

RDF makes use of the Strategy pattern, as all objects listed above have methods *toNT* and *toQuads*, meaning there's no need to test for type or feature to know whether or not they can be called.

RDF also makes use of the Composite pattern, as Collection and Formula will call on their leafs when toNT and toQuads are called.

5.12 RDF Loader

Branch	Graphite
Location	`graphite/rdfloader.js`
Dependencies	Loader (page 63) , RDF Parser (page 68) , Utils (page 72)
Size:	4.0 kB
SLOC	50
Design Pattern	Facade (page 35)
Test result	N/A

The RDF Loader module is a very simple module, and does in fact only consist of a single function. The function takes an IRI that can be dereferenced as a graph, the name of that graph, and a function to call when it's loaded. What it passes along is the graph that's been fetched, ready for further processing.

The module acts as facade for the underlying modules that sports a much greater API, and delivers a single, easy-to-use function.

5.13 RDF Parser

Branch	Graphite
Location	`graphite/parser.js`
Dependencies	JSON-LD (page 69) , RDF JSON (page 69) , RDF/XML (page 70) , Turtle (page 70) , Utils (page 72)
Size:	1.4 kB
SLOC	34
Design Pattern	Strategy (page 39)
Test result	5 tests, 12 assertions; Total average: 168, Average/assertion: 14

The RDF Parser module enables us to parse RDF independent of its serialization. For now it needs to be configured by the user to let it know which parser to use, but the goal is to make it detect the serialization on its own.

The module has been designed with the Strategy pattern in mind. By treating all parsers as different strategies to parse RDF, it enables us to easily add and remove additional parsers in the future (e.g. if we want to be able to parse RDFa).

Figure 5.6 show the dependencies within the RDF Parser modules (RDF, Loader, Promise, URI, and Utils being included to show external dependencies).

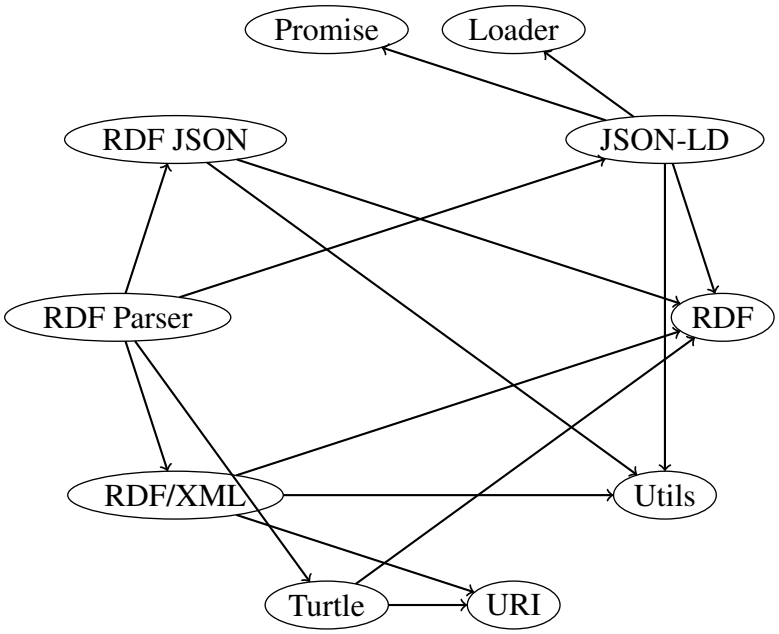

Figure 5.6: Dependencies between the modules related to the RDF Parser in Graphite.

5.13.1 JSON-LD

Branch	Graphite
Location	`graphite/parser/jsonld.js`
Dependencies	Loader (page 63) , Promise (page 71) , RDF (page 67) , Utils (page 72)
Size:	17.5 kB
SLOC	361
Design Pattern	Strategy (page 39)
Test result	22 tests, 63 assertions; Total average: 249, Average/assertion: 4

The JSON-LD module parses JSON-LD into RDF, and partakes in the the Strategy pattern by being a ConcreteStrategy. It also makes use of the Loader and Promise modules as it supports dereferencing URLs that are used in @context.

5.13.2 RDF JSON

Branch	Graphite
Location	`graphite/parser/rdfjson.js`
Dependencies	RDF (page 67) , Utils (page 72)
Size:	1.5 kB
SLOC	39
Design Pattern	Strategy (page 39)
Test result	8 tests, 9 assertions; Total average: 923, Average/assertion: 103

The RDF JSON module parses RDF JSON. It participates in the Strategy pattern as a Con-

creteStrategy.

5.13.3 RDF/XML

Branch	RDFQuery
Location	`rdfquery/parser/rdfxml.js`
Dependencies	RDF (page 67) , URI (page 71) , Utils (page 72)
Size:	14.7 kB
SLOC	264
Design Pattern	Strategy (page 39)
Test result	29 tests, 136 assertions; Total average: 1785, Average/assertion: 13

The RDF/XML module is taken from the library RDFQuery, and supports about 60% of the tests given in the official RDF/XML test suite[6] (136/169 of good tests passes; all 56 bad tests fails).

The modules participates as a ConcreteStrategy in the Strategy Pattern, along with all the other RDF parsers and the RDF Parser module.

5.13.4 Turtle

Branch	RDFQuery
Location	`rdfquery/parser/turtle.js`
Dependencies	RDF (page 67) , URI (page 71)
Size:	16.9 kB
SLOC	474
Design Pattern	Strategy (page 39)
Test result	4 tests, 31 assertions; Total average: 315, Average/assertion: 10

The Turtle module originates from RDFQuery, and supports all of the tests given by the Turtle Test Suite`http://www.w3.org/TeamSubmission/turtle/tests/`.

[6] `http://www.w3.org/TR/rdf-testcases/`

5.14 Promise

Branch	Graphite
Location	`graphite/promise.js`
Dependencies	None
Size:	25.1 kB
SLOC	279
Design Pattern	None
Test result	Not available

The Promise module is an integration of the When library. It implements the Promise pattern (section 2.2.6.2), which gives us additional tools to handle asynchronous calls.

5.15 Tree Utils

Branch	RDFStore
Location	`rdfstore/utils.js`
Dependencies	B-Tree (page 55)
Size:	13.9 kB
SLOC	380
Design Pattern	None
Test result	2 tests, 4 assertions; Total average: 13, Average/assertion: 3

The Tree Utils sports several handy functions used by many of the components originating from RDFStore. It also contains an implementation of a B+ tree, which is used by the Engine.

5.16 URI

Branch	RDFQuery
Location	`rdfquery/uri.js`
Dependencies	Utils (page 72)
Size:	9.5 kB
SLOC	179
Design Pattern	None
Test result	74 tests, 99 assertions; Total average: 230, Average/assertion: 2

The URI module originates from the RDFQuery project, and handles many utility-functions used when working with IRI. It applies no SDPs as we can see.

5.17 Utils

Branch	Graphite
Location	`graphite/utils.js`
Dependencies	Lexicon (page 62)
Size:	26.8 kB
SLOC	498
Design Pattern	None
Test result	37 tests, 141 assertions; Total average: 130, Average/assertion: 1

The Utils module is a collection of utility functions used throughout the framework. A lot of the functions originates from the Underscore project. It doesn't apply any SDPs.

Chapter 6

Discussion

[TBW]

6.1 Modules

[TBW]

6.1.1 Abstract Query Tree

[TBW]

6.1.2 Engine

[TBW]

6.1.3 SPARQL

[TBW]

6.2 Use of 3rd party libraries

[TBW]

6.3 Entailment

[TBW]

6.4 Serverside implementation

[TBW]

6.5 XDomainRequest (XDR)

[TBW]

6.6 Asynchronous Module Definition (AMD)

[TBW]

6.7 Test-driven development (TDD)

[TBW]

6.8 Other patterns of design

[TBW]

6.8.1 Lazy Loading

[TBW]

6.8.2 Observer

[TBW]

6.8.3 Representational State Transfer (REST)

[TBW]

6.8.4 Architectural Styles

[TBW; including reference to Roy Fieldings dissertation "Architectural Styles and the Design of Network-based Software Architectures" `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`]

Chapter 7

Conclusion

[TBW]

7.1 Further Work

[TBW]

Codebase

As the codebase for Graphite is rather large (approximately 36 000 SLOC, or between 500 and 1000 pages, depending on the format you print it in), we've decided to just refer to the repository at GH.

The complete framework, with all tests and demoes, is available at `https://github.com/megoth/graphitejs`.