Methodically selecting a test framework for REST

A literature review

Timmy Olof Olsson
Abstract

Representational state transfer (REST), is an architecture style for network related software. It was proposed by Roy Thomas Fielding as his Ph.D. dissertation in the year 2000. It intended to improve scalability and simplify the designing. In 2016, the Swedish tax agency (Skatteverket) has begun replacing old services related to their information system (BasInfo) with RESTful web services. This requires rigorous testing, planning, and evaluation to assure it works. At the same time, they adapted agile as their development process, which added additional requirements for how the software is going to be tested. The goal of this thesis is to provide research about methods and frameworks that are meant for testing REST. It includes defining what functionalities a test framework should have, and how it stands in regards to the agile practice. We will conduct a literature review, which will describe the methods and frameworks, by how they work, their purpose and an analysis of what functionalities are relied on. They are evaluated based on their drawbacks and advantages in regards to agile view on time, quality and learning curve. From our research, we discovered there was two type of methods and frameworks. It indicated that a framework should have the capabilities of a general-purpose programming language, support generation of test cases and test data. Furthermore, it also showed potential in the aspect of agile’s view on time, quality and learning curve.
Acknowledgements

I would first like to thank my supervisors (associate professor) Anders Broberg and (system developer at Skatteverket) Rikard Skelander. Anders, for having the time on a weekly basis discuss the thesis progress and issues, and Rikard for the technical discussions about testing. Secondly, I would also like to thank (section manager at Skatteverket) Christina Svedberg, for giving me this opportunity and lending me an office to work at and other essential equipment. I’m also thankful for the support of the employees of Skatteverket, and their interest in my thesis.

Finally, I would like to express my thanks to my family and friends, for always being supportive and wishing me the best of luck.
# Contents

1 Introduction 1
   1.1 Problem statement 1
   1.2 Purpose 2
   1.3 Thesis outline 2

2 Background 3
   2.1 Agile 3
   2.2 Testing in general 4

3 Method 5
   3.1 Research 5
      3.1.1 REST 5
      3.1.2 Agile 5
   3.2 Database 5
   3.3 Data analysis 6
      3.3.1 REST 6
      3.3.2 Agile 6

4 REST 9
   4.1 What is REST 9
   4.2 Constraints 9
      4.2.1 Client-server 9
      4.2.2 Stateless 10
      4.2.3 Cache 10
      4.2.4 Uniform interface 10
      4.2.5 Layered System 10
      4.2.6 Code-on-Demand 10
   4.3 Concepts 10
      4.3.1 Resource 11
6.4.2 Testing stubs framework
6.4.3 Connectedness and exploratory testing
6.4.4 Model based testing
6.5 Conclusion

7 Conclusion
7.1 Research about testing REST
7.2 Functionality for test frameworks
7.3 Agile aspect

8 Summary and Future Work
8.1 Summary
8.2 Future work
1 Introduction

Representational State Transfer (REST) is an architectural style, based on principles and constraints, for designing a distributed hypermedia systems[1]. REST has seen a massive gain in popularity within the latest decade in the industry. A significant contributing factor being it is more lightweight than the popular web service Simple Object Access Protocol(SOAP), and its learning curve isn’t as steep1.

Even though REST has these benefits, it comes with the problems caused by its distributiveness and loose coupling, when testing the functionality of service. The trust for the web service always being available and working when needed, as stated below:

“Web services are not yet widely used because of security concerns. But there’s an even bigger roadblock waiting just down the road – it’s called trust. The big issue is ‘Will the service work correctly every time when I need it?’”

CBDi Forum, 2002

This is the most prominent challenge of a web service. It is especially true for organizations that provide a service that is vital for other parties to always be available, and this is why tests are developed to give this assurance of trust. Designing some of these tests requires that the tools have the capability and functionality to support the operations of a test case, and can provide the information needed to validate the result.

This paper will present a loose interpretation of the REST architectural style, including formal definitions about some of the technical aspects of REST. Furthermore it will present some of the current testing techniques that is aimed for RESTful web services. However, it should be noted that the research on this topic is limited, but some articles have surfaced with different approaches to testing REST. The paper will also try to add the perspective from an Agile point of view, on the subject of testing and the benefits the tests can have on the value of software.

1.1 Problem statement

The Swedish Tax Agency (Skatteverket) has requested for a pilot study for the latest (or known) research about testing REST web services. Coupled with what features or criteria is recommended for a test framework to have when developing tests for new REST services. This study will also try to include a perspective from the Agile standpoint, to get a sense of how it can help in an Agile environment.

Based on the given problem, the questions for the paper to cover, are expressed as follows:

---

RQ1: What research is there about methods and test framework, that is within functional testing, for RESTful web services.

RQ2: What functionalities should a test framework have. Based on the findings about the test methods and test framework from the previous question.

RQ3: How would the defined functionalities from the previous question, assist an agile team?

1.2 Purpose

The purpose of this paper is to provide a set of criteria that is based on research about testing of RESTful Web services. The criteria can act as a guideline for what is needed in a tool when needing to test REST specific abilities. Even though REST share the common problems as other web services, its constraint are important factors to take into consideration.

1.3 Thesis outline

The following is how the thesis structure and organized:

Chapter 2: Background Presents the underlying reason to why this thesis was created, and give a short introduction to REST, agile and testing.

Chapter 3: Method Introduces the approach for the literature review. It defines the protocol for what the inclusion criteria are, and how the research was analyzed.

Chapter 4: REST This chapter will provide a background for how REST came to be. Then an overview of the architecture, and its constraints.

Chapter 5: RESTful Web Services Testing Here, we will present test frameworks and test methods for RESTful web services, which will include an analysis of the features.

Chapter 6: Agile testing In this chapter, we introduce the concepts about agile and testing, after that, we present an evaluation of the frameworks and methods in regards to the concepts.

Chapter 7: Conclusion Draw our conclusions

Chapter 8: Summary and Future Work A summary of the approach and result, and a mention of what was not included in the thesis. Nevertheless, it could be interesting to have in future work.
2 Background

In 2016 the Swedish Tax Agency (Skatteverket) in Umeå was given the responsibility of maintaining a legacy system called BasInfo. BasInfo is a critical part of the organization; it internally provides services for other systems within the organization that other agencies utilize. The system collects and manages sensitive personal information of people living and working in Sweden, thus its a very central part of the organization.

Skatteverket, previously Riksskatteverket, at Solna, began the development of BasInfo in the 90s. The system was written in pure C, with little to no documentation which also increased the complexity of maintaining it during the decades. Finally, Skatteverket set out to replace part of BasInfo’s systems with a more reliable and maintainable “newer” technology REST. REST is the architectural style for web services that have gained high popularity in the industry, since the publication of Roy T. Fielding’s dissertation of REST back in the year 2000.

Being a tax agency, developing a new system to store sensitive and vital information, demands for a high set of standards in regards to reliability and functionality. Thus, testing to ensure that the service is working correctly, is crucial for a scenario like this. With the growth of popularity for web services in the industry, the number of tools for testing web services has also increased. Each tool has its features and drawbacks, and deciding on a tool for specific needs has become harder. Therefore having a set of criteria that can weigh in on the decision can be helpful.

2.1 Agile

Agile is a development practice, with the purpose on developing quality software, through testing and collaboration. It defines a set of roles e.g., team lead, team member, product owner and stakeholders. Which all have their own responsibility in order to deliver high quality software seamlessly. The Agile practice emerged as a manifesto in the early 2000s, as a response to the issues of traditional software development approaches\(^1\).

It caused company after company to and adapt this practices, and eventually it became the standard of the industry. Skatteverket, recently, adapted the practice, in order to conform itself the same standards as the rest of the industry.

\(^1\)http://agilemanifesto.org/history.html [Accessed 22 May 2018]
2.2 Testing in general

In agile, testing is fundamental in order to achieve quality software. The purpose of testing is confirming if a behavior of a software conforms to the expected behavior. However, a challenge with testing is knowing whether if these tests can assure the value and quality of a software. In agile, there needs to be a definition of "done", in order to say whether the tests are sufficient enough or not.
3 Method

The approach of this thesis is to apply for a general literature review. The objective of a literature review is to derive answers to the research questions, through summary, and analysis of research related to the subject.

3.1 Research

This literature review will take a look at research that is about methods and frameworks, which are related to testing REST. The resource will then continue, to focus on agile and testing, and evaluate the previous findings based on what is found on this topic. This section will define the inclusion criteria for literature to be used in this thesis.

3.1.1 REST

The research about REST architecture will be for the majority, based on the original dissertation. For this thesis, it is required that the research about the methods or frameworks are about functional testing. Articles about, e.g., performance tests, security tests, endurance test, etc., etc. are excluded, as they are non-functional tests.

The following keywords were used research: RESTful, Web service, functional testing, test frameworks, test methods.

3.1.2 Agile

The concepts that are related to REST, will include literature and blog comments by authors that are known and cited over 100 times. Furthermore, it will include websites that are authored by known companies (Microsoft and Amazon). This is to gain a general overview of what aspects of agile and testing are important.

Research that connects agile and the findings from RQ2, are articles that only stems from the given database. Which provide what drawback and advantages the method or framework feature have, in time, effort and quality.

The following keywords were used: Agile testing, Agile continuous delivery,

3.2 Database

The following list, identifies what databases are used for finding the research:

- Google Scholar
3.3 Data analysis

The methodology of analyzing the findings are different for REST and agile. Since they are two different subjects, they have separate approaches to analysis.

3.3.1 REST

In this thesis, we are going to describe and analyze two different frameworks and methods that are meant for testing RESTful web services. The analysis will be to look for features, which are either directly presented or found by implications in the research articles. A feature is a functionality that is required to perform a specific task. The primary objective of this analysis is to shape a list of functionalities for a test framework.

Test Frameworks

The article *Test-the-REST: An Approach to Testing RESTful Web-Services* authored by Sujit Chakrabarti and Prashant Kumar (2009), is a unit test framework. The article includes details about the implementation, architecture, test cases and validation of responses.

The second framework, presented in the article *A FRAMEWORK FOR TESTING RESTful WEB SERVICES* authored by Hassan Reza and David Van Gilst (2010). It presents a specification for a stub framework and not an implementation. It includes a specification for responses, interface, and validation of requests.

For the unit test framework, the focus will be to describe and analyze the features of the architecture, test cases and validation of responses. While the stub test framework, the attention is directed towards describing and analyzing the features of the interface, generated responses and validation of requests.

The features that are found are then to be evaluated if they are a necessity or can be excluded from the list of functionalities.

Test method

Beyond describing the methods, the analysis is to find algorithms used in the methods. It is to implicate what functionalities are relied upon. Furthermore, operations that are used in order to achieve a task are also considered as a functionality.

3.3.2 Agile

The analysis will consist of what advantages, and as well the disadvantages, which are based on articles, that are related to the features of methods and frameworks. This is to be later
evaluated to what it can be beneficial for agile in time, effort and quality.
4 REST

This chapter gives a brief overview of what REST is and what derived the architectural style and its concepts. It’s my interpretation of Roy Fielding’s dissertation "Architectural Styles and the Design of Network-based Software Architectures", 2000 combined with other sources to create a more breadth analysis.

4.1 What is REST

REST, Representational State Transfer, is a proposal for an architectural style for hypertext systems. Roy Thomas Fielding suggested it in the year 2000 as his doctoral dissertation. Furthermore, according to Google Scholar, it has since then been cited over 6000 times. The dissertation has then brought the inspiration to various implementation across multiple programming languages.

The motivation of REST was to create a framework for how the web is supposed to work and provide guidelines for the standard of the web. With the rapid development of different web features in the early 90s, that complemented the lack of functionality of current web architecture. As a result, it incited the development of the HTTP/1.0 and 1.1, and from there REST became the design style for communication[1].

REST's design is based on a set of constraints that are based on engineering software principles, which are placed on the components.

4.2 Constraints

Roy T. Fielding designed REST by a list of constraints on the components, which derived from the design styles and a set of principles it tries to maintain. REST emphasizes on "scalability of component interactions, the generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems."[1]. This is a loose interpretations of the constraints, based on Roy T. Fielding’s dissertation

4.2.1 Client-server

Separation of concerns principle adds a constraint on the client-server. What this does is separating the user interface from the server and moving it to the client. It allows the components of the server and client becoming more loosely coupled and thus improves the scalability for the server side [1].
4.2.2 Stateless

Client-stateless server style applies the constraint that a single request from the client to
the server should be sufficient enough for the server to understand the request. It means
that the client can’t store a context, and perceive that the server can determine a response
based on previous requests. The responsibility falls on the client to keep track of its current
state[1]. According to Fielding, this should give the properties of visibility, reliability, and
scalability.

4.2.3 Cache

Client-cache-stateless-server style imposes the constraint that the if the client should deem
necessary to cache the data of a message, its required that the request label the data as a
cache. So the server does not need to redo the process of creating the response if the client
is sending a similar request. It improves the network performance, by lowering the time for
the server to respond[1]. This constraint is, as stated by Roy, optional.

4.2.4 Uniform interface

The Principle of software engineering generality emphasizes on ”designing software that
is free from unnatural restriction”[5]. It is a very critical part of REST style architecture,
the components of the architecture should conform to an interface that provides a simple
interaction between the components. It gives the possibility for each component to evolve
separately from each other, and deploy independently from each other. The benefits are
less complexity so that the system is easier scaled and improves visibility[1]. The con-
straint is composed of four separate constraints, identification of a resource, manipulation
of a resource, self-descriptive messages, hypermedia as the engine of Application state, the
constraints will be explained later in this chapter.

4.2.5 Layered System

Layer system style creates the constraint that there is a hierarchy of layers. It does not allow
a layer to have any coupled relation with a layer that’s beyond its intermediate layer, and it
gives the possibility of encapsulating outdated services and replacing them[1].

4.2.6 Code-on-Demand

Code-on-demand-style is an optional constraint which allows the client to download ex-
tended functionality after deployment. However, it affects the visibility, and thus it is a
voluntary restraint[1].

4.3 Concepts

In the architectural style of REST, a distinguishable constraint is a uniform constraint. As
mentioned earlier, the uniform interface relies on four other constraints. Roy T. Fielding
discussed those constraints in his dissertation. They are in place to achieve an interface that
is simple and aligned with the principle of software engineering generality.

4.3.1 Resource
A fundamental concept of REST is the resource. It is described as a stream of bits that can be, e.g., a document, audio, video file, etc., etc. Also, for every resource, there’s a unique identifier. With the identifier, it gives the possibility for a client to access a specific resource and manipulate it. A resource is mapped to a set of entities that either represents or identifies a resource. There’s no requirement for the REST resource to point to a nonempty entity.

4.3.2 Identification of a resource
To be able to differentiate each resource, a unique identifier is required. In the beginning, REST used UDI (Universal Document Identifier), later changed to URI (Uniform Resource Identifier). Defined in [2] it defines URI as:

‘A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource.”

This could be interpreted that URI just provides the identification of the resource. It is however incorrect because URI can both be used as a locator and identifier, as stated later in [2]:

‘A URI can be further classified as a locator, a name, or both. The term ”Uniform Resource Locator” (URL) refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism (e.g., its network ”location”).”

It is necessary to say, that URI can provide the mechanism for access to the resource, but it is not always the case[4]. URLs primary goal is to act as a locator of the resource, as defined in RFC 1738: Uniform Resource Locators (URL) by T. Berners-Lee, et.al.

‘URLs are used to ‘locate’ resources, by providing an abstract identification of the resource location.”

URL relies on URI for being a reference to the location[4]. Observing different documentation and implementation of REST web services, URI and URL seem to be used interchangeably, even though they are technically different.

4.3.3 Manipulation of resource
As Roy T. Fielding states in his dissertation, a representation is a need for the components of REST to communicate. A representation captures a state of a resource or changes it and transfer that state between the components.
Roy T. Fielding defines a representation as:

> ‘... a sequence of bytes, plus representation metadata to describe those bytes. Other commonly used but less precise names for a representation include: document, file, and HTTP message entity, instance, or variant.’

For an actual REST web service implementation, a representation of a resource could be, e.g., a JSON, XML or plain text that holds the current data(state) of a resource. Manipulating the state of the resource is done by sending the representation to the server, to either update/create/remove a resource.

To access the resource and interact with it, most REST web services support the conventional methods and usage of the HTTP protocol[3] as shown below with a short description of its operation.

<table>
<thead>
<tr>
<th>HTTP verb</th>
<th>Operation</th>
<th>General description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Read</td>
<td>Is used for fetching a specific resource, if unique identifier is provided</td>
</tr>
<tr>
<td>POST</td>
<td>Create</td>
<td>Create a new entity for a resource on the server</td>
</tr>
<tr>
<td>PUT</td>
<td>Update/Modify</td>
<td>Append or update current resource</td>
</tr>
<tr>
<td>DELETE</td>
<td>Remove</td>
<td>Delete a resource</td>
</tr>
</tbody>
</table>

Table 1: Short description of each http method (also known as CRUD), interpreted from [3]

### 4.3.4 Self-descriptive messages

This goes back to section 5.2, about being stateless. The interaction between a client and server should be stateless, meaning that a request should contain all the information that is necessary for the server to determine a response. It adheres to the same standards of the HTTP protocols being stateless, and gives the following properties as follow:

- Visibility - It is improved because a server does not need to know anything beyond the request itself to understand it.
- Reliability - Its reliable, because it makes it easier to recover from partial failures.
- Scalability - When having that the server does not store the state of each client, it simplifies the design and makes it easier to scale.

### 4.3.5 Hypermedia as the engine of Application State (HATEOES)

In Roy T. Fielding’s dissertation, when explaining the uniform interface, he mentions hypermedia as the engine of application state as one of the four interface constraints of REST architecture.

The term hypermedia derived from Hypertext. Defined by Ted Nelson, hypertext is text that has a link to other texts. Hypermedia is a complement to hypertext; it can have text but also, e.g., music, video, and graphics[6].
In this case, it is used as a way to help the client to interact with the navigation of the resources dynamically. By providing a structured response to the client of what resources exist in URI format, and a representation of those in URI format, it controls how the user explores and manipulates those resources[4]. Thus it is becoming the 'engine' of the application state.

Even though being a vital element of the REST architecture, it is a common constraint that isn’t followed by different APIs claiming to be RESTful according to a blog post “REST APIs must be hypertext-driven” by Roy Fielding’s in 2008.

4.4 RESTful

As explained in the earlier section, Roy T. Fielding’s dissertation is an architectural design for how the web is supposed to work, according to Fielding. Seeing that REST is not defined a defined protocol like SOAP, leaves much room for interpretation of how REST should be implemented in practice. This coined the term ‘RESTful,’ which is meant to describe if an web service implementation of REST "really" adopt the constraints[3].

4.5 Conclusion

The architecture is derived from engineering principles, and for good reasons. However, based on the concepts the following list identifies some of the challenges of testing.

- **Stateless:** The issue with a stateless interaction between a server and client, is that the client needs to keep track of its own state. This is a challenge when testing, as every tests can be interpreted as a its own client. It can therefore, become an issue when testing a functionality that depends on previous interactions.

- **Performance:** Guaranteeing the system is always available, and responsive within a time interval. Is a common issue when testing, as it requires control of the system and its environment in order to monitor and evaluate the situation.

- **Validating interaction:** Since a request or response, in an interaction, consists of multiple parts eg., parameters, body and HTTP operations. Since an interaction consisting of multiple parts, that are different from each other. It creates the challenge of having different verifications for the requests and responses e.g., parameters, media type and HTTP operation.

- **Availability:** Since resources are supposed to be mapped to each other, it becomes an issue of verifying that all resources are mapped to each other.

- **Control of the system:** A prevalent problem for testing these kind of systems, are tests that are dependent on other parts of the system, that is not always available for the tester. E.g testing the behavior of a client based on the responses from server, but the server might not be available in a testing environment.

- **Pinpoint bugs:** Tests are required to both be done internally and externally of the system. An issue is with external tests, are however, it doesn’t have the same access
to monitor the internal state of the system. It can therefore be harder to pinpoint the exact location of error.

- **Extensive testing:** As the system becomes more complicated it affects how thorough the test can be manually created. For the purpose of verifying the behavior and functionalities.

- **No user-interface:** Since REST is an API, it does not have a user-interface, consequently so it is therefore difficult to test it manual. As the requests requires specific formats and procedures in order to interact with the API. E.g, the tester (guru), is testing a GET operation, it would require the tester to manually format a proper URL with the correct parameters. Doing so, multiple times increases the chances of incorrect URL formatting. Furthermore, the tester also needs to have knowledge in how to interpret a HTTP response.

The issues presented above, are some of the challenges that are faced when testing REST services. In the next chapter, we will present frameworks and test methods that can overcome the issues related to stateless, validating interaction, control of the system and extensive testing. It includes examples and further explanation of the issues, and what possible solution is to be offered.
5 RESTful Web Services Testing

The previous chapter presented an overview of the REST architecture with the background about how it came to emerge, with its intention to bring an architecture that helps designing complex systems. Furthermore, the chapter introduced how the constraints worked and formed the interaction of REST.

This chapter will examine the research on the topic of testing RESTful web services, and as to what tests methods and purposed test frameworks exist. The examining of the research is essential for the aim of this thesis, as it is required to answer the research questions RQ1 and RQ2.

The chapter will conduct a literature review based on the protocol defined in the method chapter. It will begin examining research about test frameworks and test methods for REST. This includes describing their purpose, how they work and define their functionalities.

It is the functionality, that is going to define the criteria for a framework. A functionality, in this case, is defined as an operation to perform a specific task. E.g., for a test to send a request to REST, it requires the ability to use HTTP operation. In order to define the functionalities, the test framework will be looked at what functionality it provides. While for methods it is by what functionality it requires to perform a specific task.

5.1 Unit testing framework

Unit tests consist of assertion. Assertions are statements that verify the result of an operation is what to be expected. The conventional approach for web services assertion would be to, e.g., confirm the HTTP status code and the body of the response from the server.

However, as mentioned in section 4.3.5, REST sets no restriction on the media type of the body. Which in turns adds an extra layer of complexity when verifying if the media type is correct.

In an ideal world, each test is atomic. Meaning that there is no dependency between different tests, they can be executed in any order without affecting the outcome of the other tests. Unfortunately, in the case of verifying a functionality which depends on other features to be executed beforehand, it can cause the test cases to be codependent, e.g. testing a website with a login page.

The website first requires a successful login to gain access to the functionalities of the site. A test case would then need to provide authentication before testing the websites API. In the case of REST, with the constraint of statelessness, the test requires to re-authenticate itself every time before testing. For the server to go through the process of authentication several times during a testing session is inefficient and time-consuming. That said, REST provides the option of using cookies, which can act as a token for login authentication[11].
This prevents unnecessary re-authentication after a first successful login. Tests like these, which tests the API, without knowing the internal functions, are commonly known as black-box tests\(^1\).

The article authored by Chakrabarti, S., and Kumpar, P. (2009) provides an approach for how to test RESTful web services with the purpose of black-box testing. They propose a test framework called Test-The-REST (TTR) for their in-house RESTful web service.

### 5.1.1 Architecture features

The architecture support test cases with the functionality of extensible plug-in validators\[12\]. The validator is used for validating specific content of the response, e.g., the media type\[12\]. In the case of changes in the requirements regarding validation, it is useful to be able to interchange validators.

The architecture consist of a test case validator module, which is the starting point, it validates that the syntax of the test case is correct before proceeding. The test case is then passed onto their main component (HTTP engine)\[12\], that interpret and invoke the HTTP operations described in the test case.

### 5.1.2 Test case

As mentioned in the beginning of the section, REST allows for more than one type for representation of the response. Therefore, in the situation where it is not known if the response is, e.g., going to be either representation in XML or JSON. The test language provide "composite validation"\[12\], it incorporates the logical expression of "|" (or) and "&" (and). An example of composite validation of a response can be seen in the Figure 1.

```
<testcase>
  <id>TTR-COMPOSITE-VALIDATION</id>
  <URI>http://localhost/cgi-bin/hello-suji.t.cgi</URI>
  <atomicity>atomic</atomicity>
  <method>GET</method>
  <response>
    <representation mediaTypeExpression="xml\|plain text"/>
    <mediaType id="xml" value="XML"/>
    <mediaType id="plain text" value="PLAINTEXT" src="D:\ResultValidationData.txt"/>
  </response>
</testcase>
```

**Figure 1:** Test case with a composite validation - adapted from (Chakrabarti and Kumar, 2009) © [2009] IEEE

The situation described in the beginning of this section required test cases to be dependent on each other because of the sequence of execution. TTR offers a feature referred to as "composite test case"\[12\], it grants the option of allowing test cases to call other test cases during execution. TTR support any level of nested calls, it does not, however, support recursive calling\[12\].

Continuing the example of the website and login page. To avoid the over excessive use of bandwidth is to use a token instead. For it to work with several tests cases, the token needs to be passed on between the tests cases. TTR borrows the feature of using parameters and

\(^1\)https://en.wikipedia.org/wiki/Black-box_testing [Accessed 20 May 2018]
Figure 2: A composite test case - adapted from (Chakrabarti and Kumar, 2009) © [2009] IEEE

return values from the traditional programming language[12].

In the Figure 2, we can see the tags var, return, param. The variable of the test case TTR-BL-3-UC is named ‘x’, which is initiated by the output of the test case TTR-BL-1. The variable is thereafter passed as a parameter to test case TTR-BL-2.

5.2 Testing stubs framework

Extensive systems that are loosely coupled, and controlled by different stakeholders, it presents the issue of not having access or control over the system. Which in turn is challenging for developers trying to create unit tests for a component that is dependent on other parts of the system[8].

In the work of Hassan, R and David Van Gilist (2010), they try to explore an approach of how to derive a specification for a framework, which allows for creating stubs that abide by the given test data. The primary requirement is that it should be able to validate requests. It defines what services are available, verifies the contents of the request, and provide autogenerated responses.

5.2.1 Defining the stub

The purpose is to define an interface which is capable of simulating the same behavior as a RESTful web service. To do this, it should have the essential elements of a RESTful web service[8]. The simulated service must be capable of providing direct access or links to the resources, and store data from the incoming request. The interaction with the service needs to behave similarly to how RESTful web service react to HTTP operations e.g., a request with invalid URLs should be rejected and responded with a 404[8].

5.2.2 Validating interaction

Using the specification of the service, the interface, is to validate the interaction with service is a modest approach. However, as a result of the lack of restriction with the parameters in URLs it brings the issue of validating if the parameters are in, e.g., the correct order, type, and value[8]. In larger systems, where the internal communication can affect the complete behavior, validating the parameters is then a must.
An approach would be to define a specification, the framework can refer to when validating the parameter. It can, e.g., provide a pattern or document type definition (DTD), the parameters need to abide [8].

5.2.3 Generating response data

There are two conventional approaches when generating test data. Data-centered and logic centered. However, they can be combined to form a more complex response, which can be used to cover an entire test case scenario [8].

Data-centered is the approach where there exists a fixed dataset for matching against inputs, and responses based on the input data. Moreover, the fixed data is to be representative of the real world interaction in order to give appropriate testing. Logic-centered is similar to a program that has an implementation of logic that can generate the output based on the input [8].

The article gives the example of white-box testing, which is a test that tests the internal workings of a system. In this case, the system has a boundary condition, which gives two different outcomes depending on the response from an HTTP request, as shown in Listing 5.1. The authors present the solution in XML, by using a data-centered-approach, that iterate through a list of sample test case values and the expected outcome.

**Listing 5.1:** The client made in MochiKit AJAX tools, sends request to REST - adapted from Hassan, R and David (2010)

```javascript
function update_aims_temp() {
  deferred = doSimpleXMLHttpRequest("http://web/aims_temp.cgi");
  deferred.addCallback (callback);
}
function callback(Result) {
  value = Result.responseText;
  temp_display.innerHTML = value;
  if(value > 50){
    temp_display.backgroundColor = red
  } else {
    temp_display.backgroundColor = blue;
  }
}
```

In the case of a more complex scenario where the system requires a dynamic response, e.g., message length, together with a data set. Then the two approaches, logic and data-centered can be combined to form a solution. Where the data-centered is the first phase of generating a message, and the logical-centered is to calculate the response message length. As shown in the example of Listing 5.2.

**Listing 5.2:** The service template responds to the call - adapted from Hassan, R and David (2010)

```xml
<datablock>
  <head length="2, len(message)">data points="[pass = rand(1,10)]"></head>
  < datapoints >
    [ pass = 1, gen_datapoints(data points) ]
  </ datapoints>
</datablock>
```
5.3 Connectedness

The chapter about REST, presented the constraint hypermedia as the engine of application state (HATEOS), as one of the most common violated constraint the architecture style.

The constraint applied that every resource should be accessible to the client, without the client knowing how to frame the URIs. It is the responsibility of the web service to provide the complete URIs through its resource representation. For this to be possible, it requires that the resources be interlinked to each other and provide the means of accessing the other resources.

Currently, there is still little known about how to test this constraint. The authors Chakrabarti, S. and Rodriguez, R. (2010) presents an initial approach on how to check a web service for "connectedness" and the specification for how to implement such test.

Connectedness requires that all resources are accessible with successive GET request on the URL[10]. The response includes URLs to resources which are related to the current resource. If the response consists of a URL to a resource that doesn’t exist then that web service is considered not to be connected.

The lack of connectedness affects several aspects of the system. Usability, if the client is required to know how to build the URLs, the service is then limited to specific clients. Performance, it is a waste of computer power when resources exist but are unreachable and consume memory. Furthermore, the client interaction is affected if the links point to a nonexistent resource[10].

A connected web service can be illustrated as a tree, where each node represents a resource as seen in figure 3.

![Figure 3: A binary tree representation of of a booking system for an airline](image)

The Figure 3 represents a booking system for an airline, which provides access to the resource "Available" flights and the resource "booked" flights. The resource "Available" contains URL for the flights that are available for purchase. The resource "Booked" represents the booked flights and the URL to that flight information.
In the case that a flight is incorrectly listed as available and the resource provides a broken URL that leads to nowhere, then the service is not connected according to the definition. Moreover, when a new flight is published as available, but because of a bug, the resource "Available" doesn’t have the URL. Then it is also not a connected REST service.

5.3.1 Diagram representation

To illustrate the relevant aspect of the REST service that is related to the problem of connectedness. There are two type of diagrams. POST class graphs(PCG)[10] illustrated in Figure 4. It is a directed graph which represents resources as nodes and the edges as HTTP POST operation that is used to create an instance of a resource. POST object graph(POG), is a tree representation of the parent resource and its child resources[10] as seen in Figure 3.

![Figure 4: A PCG representation of booking system for the airline](image)

A resource is called the source of a POST request. If it is a resource that provided a URL to perform POST request on to create a new resources[10]. In Figure 4, we can see that a POST request from "Available," to the resource "Flight" creates a new instance of the resource. Therefore, "Available" is called the source of the POST request.

5.3.2 Testing process

The testing process begins with generating a programmable representation of the PCG diagram. The PCG is then used for generating tests or verification. As mentioned earlier, each edge in PCG represents a POST. In a programmable context, the edge is a method that provides all the necessities to send a POST request to create a resource.

The next step is using the PCG to generate test resources that are then enlisted in a POG on the client side. It is to keep track of the created resource objects on the server and to be later used in the verification process. POG is generated by an algorithm that begins with exploring the PCG level by level. Then, for every POST method a node has, it will use it to create a (random) number of resources and add them as nodes in their respective level in
the POG[10].

The following step in the process is to do a depth-first search. It starts with performing a GET request on the base resource, e.g., "Airline," and retrieve a stack of URLs. Then do a GET on the top item on the stack, and repeat the process as in-depth first. Moreover, it counts on how many nodes it has visited.

The final step is to verify that all resources that were created are reachable from the base resource. From the first step, it is known how many resources that were built, and in the previous how many were visited. Therefore we can perform a simple equality check between the numbers of generated and visited nodes. If equal, the RESTful web service is connected. Otherwise, it is not[10].

5.4 Model based testing

The introduction chapter mentions the trust in knowing that the web service is always available when needed. According to the principles of testing, tests cannot prove that a program is bug free. It can, however, show that defects exist and reduce the number of it[13]. In addition, it provide assurance of reliability. In the case of large systems with a great set of functionalities, it is impossible to develop extensive tests that tests for every type of scenarios e.g., combinatorial testing of the input parameter[13].

Model-based testing (MBT) is an alternative to the traditional way of developing tests. It uses models to describe inputs/output, sequences and represent the behavior or state of the system when it is undergoing testing. The purpose is to explore the different conditions and scenarios by auto generating test cases based on the model[15]. This could be a favorable approach when extensive testing is important. Code coverage is a term used to describe the amount of code being utilized during testing, and with auto-generating tests it achieves a higher code coverage than traditional way of testing. However, it should be noted that a high code coverage does not say how good the tests are on detecting faults[14].

Currently the subject of MBT and REST services is still rather an unexplored topic. There exist only a handful of attempts. The authors Pinheiro, Pedro Victor Pontes et al. (2013), explores the approach of using a protocol state machine, which is meant to describe the different states modeled in UML protocol state machine diagram.

5.4.1 Protocol state machine

There are two types of state machines, behavioral state machine and protocol state machine[18]. They are both concepts used for modeling a specific aspect of the system. The behavioral state machine is used for model a specific components behavior in a system. While the protocol state machine model the usage of the transition in the system based on the protocol[18].

Event protocol is used when wanting to show the sequences and responses which are related to a set of communication protocol without having to show its behavior1 (illustrated in UML) [17]. It gives the possibility to define in what order the operations can be invoked

---

3The behavior of a program being the output
without having to implement its behavior.

The Protocol state machine consists of states and transitions. The transition connects two states by their relation to the precondition. Representing the state machine as a set notation\[16\], Let \( m \) represent the state machine, then:

\[
m = (S, T, P, F)
\]  

\((5.1)\)

In the set 5.1, the notation \( S \) is the state, \( T \) the transition of the two states, \( P \) can be applied in order to create complex transitions\[16\], \( F \) the final state consist of a set of states that act as the enclosing state.

Each transition is associated with a precondition(guard) and a postcondition. This means that for every operation of a transition, there gonna be an initial condition before the transition that should be obtained and a condition once the transition is triggered\[18\]. The transition can be described as tuple\[16\], Let \( t \in T \), then:

\[
t = (s_i, e[p], s_j)
\]  

\((5.2)\)

In notation 5.2, \( s_i \) is the state occurring before state \( s_j \), the notation \( e \) represents the event being a CRUD operation (e.g., POST, UPDATE etc etc), and the precondition \( p \). As illustrated in figure 5.

**Figure 5:** Transition from a simple state i to a simple state j, with the conditions

There are two different type of states, a composite and simple state. The composite state consists of substates (i.e. simple states), while the simple state is a boolean expression used to express the resource state. Therefore there exist only two methods, \( \text{OK} \) and \( \text{NOT FOUND} \), when resources are called on. If a request e.g., \( \text{GET} \), is successful the method \( \text{OK} \) returns true if the response is of the 200 status code. If \( \text{GET} \) is unsuccessful and give the response 404, then \( \text{NOT FOUND} \) returns true\[16\].

**Figure 6:** Composite state with two simple substates

Figure 6, shows the example of a composite state when purchasing a new video game. After the transition to the composite state, it requires the postcondition \( \text{ClientGamesLibrary} \)
that the client shouldn’t have the game in its own library. The two simple states \textit{GameStoreLibrary} and \textit{ShoppingCart} together form the precondition for the next transition. The game should exist in the store library and the client has added it to the shopping cart.

Figure 7 is a behavior model of the REST service called hotel room booking (HRB), represented as a protocol state machine model. It is a system for booking rooms, it allows for reservation and cancellation of a room.

The initial state, $\bullet$, is a default state that has no transitions to it from other states. The pseudo state $\phi$, is used as a decision state, it validates the precondition from the transition of the previous state and decides the next transition. This can be seen when transition from the state \textit{WaitingForPconfirmation} to the pseudo state \textit{Choice1}, where \textit{Choice1} have to decide for one of the two outgoing transitions. The final state $\circ$, encapsulate the state of the previous states\cite{16}.

5.4.2 Generate test cases

The models are described in a abstract language (in this case XMI). The interpretor (e.i the program), convert the transitions, composite states, simple states, pseudo state and final states into programmable assets\cite{16}.

The process of generating tests cases begins with building state machine model, that describes the behavior of a specific part of the system. The model is then used for generating a direct acyclic graph (DAG)\footnote{https://en.wikipedia.org/wiki/Directed_acyclic_graph [Accessed 20 May 2018]}, as shown in Figure 8. It is used for generating the test sequences, and since it is a DAG it avoids unnecessary paths because it is acyclic\cite{16}.

A tool is then used to evaluate the expressions precondition, postconditions and the states of the model, into an abstract syntactic tree (AST)\cite{16}. The AST and the test sequences are together used to generate the actual test cases, representative in a programming language. In e.g., Java, the classes generated are Invariant Class, Events Class, Guards Class and
Figure 8: Corresponded DAG of the HRB model - adapted from Pinheiro, Pedro Victor Pontes et al. (2013)

TestSuite Class[16].

```java
@Test
public void testCaseExample() {
    if (Guards.guard1()) { // precondition
        assertEquals(true, Events.event1()); // event
        if (!Invariants.Invariant1()) { // postcondition
            fail();
        }
    } else {
        fail();
    }
}
```

Figure 9: Generated test case code example - adapted from Pinheiro, Pedro Victor Pontes et al. (2013)

Figure 9 Showcase a test case for single transition, where it simply check if the execution process holds precondition and postcondition. It is useful for detecting specific faults in the system[16].

A Short summarization of the process:

1. Create a protocol state machine model of a specific of part in system.
2. Generate a DAG of the model (as seen in figure 8), to later be used to generate test sequences of transitions.
3. Generate a representation of the conditions and the states used in the model to an abstract syntactic tree
4. Generating the java assets, the tool should generate the classes Invariants, Events, Guards and the TestSuite Class.
5. Execute the tests
5.5 Analysis

This section will compose a summary of the test frameworks and methods. It will include a list of functionalities that are identified by what framework provide, or what a method relies on in order to perform a specific task.

5.5.1 Frameworks

The functionalities of the test frameworks will be derived by what is presented in this chapter, and the motivation for that functionality.

Unit testing framework

Test-the-REST is a framework used for creating unit tests for REST services. Requirements for this framework is related to their in-house web service project. However, they present arguments to show the usefulness of the following functionalities:

- **HTTP operations**: To utilize the REST service, the test framework requires the same basic HTTP operations used by the REST service, which can be found in the Table 1 of section 4.3.3.

- **Plug-in validators**: The idea of using external validator to validate e.g., the media type. It allows the framework to be more flexible to changes.

- **Logical operators**: The test cases uses the same logical operations "|" and "\&", as used in the traditional programming languages. These operations give the ability to create more complex logical conditions than the conventional assertions, when verifying responses.

- **Functions**: The functionality of defining test cases similar to functions in traditional programming language, with variables, parameters and return values. Gives the ability for passing and retrieving data between test cases. It is useful in the scenarios of REST when passing the state of the client side between the tests.

- **Nested tests**: The framework provides the functionality of invoking other test cases within a test case. This ability give the benefit of reusing test cases. It is useful when a sequence of tests are required to be executed in a specific order.

However, the plug-in validator, is there for the purpose of being flexible and not needed to perform an actual function. Thus it can be seen as an unnecessary feature.

Testing stubs framework

This framework is used to get control over a system during testing that otherwise is unavailable. It uses stubs to simulate the behavior of a REST service. Its main purpose is to act as a placeholder during white-box testing.
• **Interface**: The stubs simulate the basic behavior of REST, by providing an interface with the basic HTTP operations, and can give access to the resources. The access can either be through directly the stub, or a direct link to the resource.

• **Predefined responses**: The predefined response is a functionality, which generate responses that is predefined to a specific request. The responses are retrieved from fixed data set. It represents real world data, and is a simple approach, but it is not sufficient enough for complex requests.

• **Auto-generated test data**: The framework have the ability to auto generate responses based on logical implementation. Which generate a response in relation to the specification of the request. It is a favorable functionality, when the responses are required to be more dynamic. Furthermore, this can be combined with a similar fixed data set used in the predefined responses, to give a the response a real world representation. However, it requires more effort than predefined responses.

• **Request validation**: The framework offers two types of validation. The first approach is by validating the request according to the interface specification to check if it follows the prerequisites before further processing of the request. The second approach is to validate the requests parameter. The parameter can be validated by its ordering, pattern and to compared to a DTD.

### 5.5.2 Testing methods

The following methods that are used for creating specific tests that is aimed towards REST. Deriving the functionalities, by looking at what functionalities are required to perform the task in the method.

**Connectedness**

Connectedness is a test method used for testing the REST constraint HATEOS. As described in section 4.3.5, it enforces the idea that resources should be accessible through each other. This test simulates the perspective from the client side trying to visit all the resources from the base resource.

• **HTTP operations**: The technique requires to be able to use the HTTP operation GET and POST. The operation GET is used to verify if the resource is available, and POST to setup resources on the server.

• **General-purpose language**: This method relies on the depth-first search algorithm. Reviewing its pseudo code as found in Listing 5.3. It is found that, the algorithm, relies on functions, parameters, variables, lists (or stacks), iterators, if conditions and logical expressions. Hence, this algorithm requires a framework with the capability of a general-purpose language, e.g., C or Java. Furthermore, the creating of the binary tree (POG), requires similar functionalities.

```
Listing 5.3: Depth-first algorithm - adapted from wikipedia

1  procedure DFS(G,v):
2      label v as discovered
3      for all edges from v to w in G.adjacentEdges(v) do
4          if vertex w is not labeled as discovered then
5              recursively call DFS(G,w)
```
• **Diagram converter**: This method relies on generating resources and test cases based on the PCG diagram. Therefore, the framework requires a feature to convert the diagram into a programmable representation. The PCG is described in a diagram, which is an abstract language.

**Model-based testing**

Model-based testing, is the method of using an abstract language to represent models, which is then used for generating tests.

- **Abstract language**: The state machine consist of states and transitions. Therefore the abstract language needs to define the terminology to describe the simple states, composite states, final states, transition, precondition and postcondition.

- **Interpreter**: The functionality of the interpreter (e.g., the tool), is to interpret the DAG and AST generate into test cases. The interpreter needs to be able to convert transitions, precondition, postcondition, states (simple and composite) into a representation of general-purpose language for the test cases.

- **HTTP operations**: The test cases need to have the functionality to interact with the service through the CRUD operation.

**5.6 Conclusion**

In this chapter we have presented a selection of test frameworks and test methods, which have then been analyzed in order to understand the underlying functionalities. From the previous chapter, we listed a few challenges that are related to REST API or in general to testing, in the following list we have concluded what framework or method can solve those challenges.

- **Stateless**: The unit test framework is equipped with parameterized unit tests, which can allow for passing the state between the tests. Without having to involve third party scripts for passing the data.

- **Validating interaction**: This challenge, consisted of verifying the different parts of request or response in an interaction. What we came to find is that, unit test framework and stubs test framework have different functionalities for validating. The unit framework, provided plug-in validator combined with logical operators, to establish if media in the response was of a specific set of types. While stubs framework confirmed requests by the interface, and parameters of the request.

- **Availability**: This is a challenge, based on the constraint that resources should be interlinked to each other, in order to provide seamless transition between resources. The method of connectedness, provided an approach to verify if all resources are reachable from the base resource.

- **Control of the system**: Stubs framework, with the possibility of creating simulated RESTful services, can be used for gaining control over parts that are not available in a testing environment.
- **Extensive testing:** Unit testing framework, with its parametrized unit tests can, if combined with a combinator, can explore the paths in a system and therefore improve the code coverage. Stubs framework with its auto-generated test data for responses, if adequate, it can increase the code coverage better than manual generated test data. Connectedness, is an exploratory testing type. It successive iterate through the resources. It can therefore as well contribute to extensive testing. Model-based testing achieves a higher code coverage by its extensive auto-generated test cases.
6  Agile testing

The previous chapter presented various frameworks and test methods. The purpose was to examine the functionalities that were relied upon, and convey an idea for a set of criteria for tools and test frameworks.

The Agile practice of developing new software for the future has become the standard of the industry. Mainly due to the fact it addresses the issues of the software development process, its cost and quality. The practice includes set of guidelines for testing practices and priorities of what aspect of testing is essential.

The objective of this chapter is to examine how the methods and frameworks with their functionalities, presented earlier, can assist in an agile environment. This chapter will begin explaining the core concepts of testing in agile, and then conduct a literature review. It will explore in what ways the functionalities provided from the previous chapter, can be used and benefit testing in agile.

6.1  Agile development

The Agile development is an iterative software development process. It has development life cycles between 1 - 4 weeks, commonly referred to as ”sprints”[22]. The Agile approach focuses on delivering high-quality software within a given time, to bring the most value to the business that meets the customer requirements. However, the dilemma here is creating value for the customer by being ”agile”, and at the same time follow the plan[43].

The triangle as seen in Figure 10, offers a different perspective on the priorities for the team. It convey the idea that through quality. Speed and continuous delivery are achievable within the constraints (cost, schedule, scope). Furthermore, its more valuable to deliver a releasable product faster, than implementing features that isn’t used.[43].

![Figure 10: The agile triangle - adapted from [43]](image)
Testing in agile relies on the method of continuous integration. Which consists of several tests phases[22]. Furthermore, a popular alternative to continuous integration is continuous delivery. Which strives towards having a deployable version of the software ready, whenever needed[24].

6.1.1 Continuous delivery

It is a process which allows the developers to release new software at any time, reliably, predictably and faster[25]. More commonly it is described as a pipeline, which consists of steps, that are tied together in a specific order. The pipeline consists of the stages commit, integration tests, acceptance tests, performance tests (optional) and deployment. Whenever something new is about to be deployed to the production, it needs to go through the whole pipeline without failing. This is to ensure the the product is stable and reliable[25][26].Moreover, it also makes it easier to identify at what stage in the process the issue occurred.

Commit

The purpose of this stage is to compile the software and then execute all unit tests, so it is ready to be move on to the next stage. This is all done together with a version controller that can store and keep track of the build. It is integral in the pipeline, as it manages and stores the project. It saves all the versions of the files, and give the option to revisit earlier changes and reapply them to the current version if needed to[25][26].

Integration tests

In this stage, the integration test initiate and set up an environment that is either real or replicated of the external systems the tests run against. The purpose of the tests is to see if the new software is compatible with the system.

Then later in the process, a collection of tests are executed against the system. It is usually automation of either all or selected tests. They are run multiple times to determine if it is reliable under pressure [25]

Acceptance tests

Acceptance tests are tests to see if features that are implemented complies with the customer requirements. It is therefore recommended to derive these tests together with the customer
directly. Moreover, it would also minimize the risk of misunderstanding of the criteria given by the customer[25].

Acceptance tests are similar to integration testing, but it has a more focus on the characteristics of the application, e.g., security, capacity, availability, and so on. Therefore, the best option would be to run the test in a production-like environment[25].

**Performance test**

This stage is optional in the continuous delivery process[25]. Its purpose is to find out how the system behaves during heavy load, and gather viable information that might reveal bottlenecks in the system[20].

### 6.1.2 Testing practices

There are several approaches for developing tests in Agile. However, the more common test practices in agile are, Test-Driven Development, Behavior-Driven development and exploratory testing[29][30][32].

**Test-Driven Development**

Test-Driven Development (TDD) is a testing practice in agile where tests are implemented before the features.

It can be seen as a cycle consisting of three steps, implement the tests, develop the feature which passes the test then refactor the code.

The approach begins with dividing the functionality into smaller segments. Test cases are then developed for each of these segments. The next part is to implement code that can pass the tests. If passed, the implementation should undergo the refactoring step, and this is to make it more into a cleaner solution. Removing unnecessary operations and conditions that add to the complexity, should improve the quality of the code and contribute to a cleaner design [29].

**Behavior Driven Development**

Behavior Driven Development (BDD), have a similar approach as TDD, it begins with implementing test cases before implementing the features.

BDD is an evolution of TDD, as TDD put more effort on verifying the system functionalities, while BDD focuses more on verifying that the features fit the desired behavior defined by the customer. TDD comes with issues of being limited to a specific programming language, which makes it harder for a for third parties to understand the test cases. Moreover, the test cases are more tightly coupled with the system features[30].

BDD tackle this by defining a language, with the intention of using it as a standard language that can be used, not only internally, but with the customer also. As it would help the developers to define the system behavior closer to the customer demands, if they were to
speak the same language. The terms used in the language needs to be sufficient enough to define and describe the user stories and scenarios[30].

The approach is to begin defining the expected behaviors that are easy to define. Thereafter, derive what the outcomes are into a set of program-traditional functionalities, that are then evolved into test cases.

Additionally the BDD approach can also be suitable for auto-generating tests. As it can help finding irregularities or issues with defined requirements[31].

**Exploratory testing**

Exploratory testing is a practice to find missing or incorrect test cases based on the user stories. It assists with evaluating if a feature is necessary to have before wasting valuable time by implementing it. The idea is to follow a set of steps designed in the test plan and instruction of how to invoke the tests. Exploratory testing should be short cycles and driven by feedback[32].

### 6.2 Agile principles for testers

In [20], we find ten important principles for testers to abide by when developing tests for features. These are all based on the agile philosophy of testing. They are meant to help to describe some of the important properties for adding value to the business by testing and what responsibility the tester have. The following list is an interpretation which describes the principles that are specifically aimed towards testing itself:

- **Continues feedback**: The testers responsibility is to help define the customer requirements, that are then implemented into tests. The tests are then run frequently, and provide useful feedback to guide the team members development[20]

- **Deliver value to the customer**: The main focus for implementing functionalities, is that it should add value to the product. Extra features that are not part of the primary requirements, does not add value and should be disregarded until the main functionalities are done.
  
  This also similarly applies to tester, the first objective is to design tests that can confirm the "happy path". The second objective is to create tests that explores paths outside the "happy path"[20].

- **Respond to change**: A common problem for testers in an agile environment, is the inevitable possibility, that customers rapidly make changes to the priorities of the stories in the development phase. A scenario like that adds extra labor and pressure on the tester to maintain the tests and adapt them the new demands. However, this should be expected and planned for.
6.3 Technical debt

Technical debt is a concept, coined by War Cunningham. It is meant to describe a project, which is borrowing time form the future, by investing less time on quality and design. In turn, the decisions to save time will eventually lead to an increase of cost and time needed to maintain the code.[25]

The main method to counter this, is by refactoring the code. Make it simpler and more efficient, with the aim of optimizing the design[25]. However, debt is necessary in the scenarios where the project needs to achieve short-term goals, but then a planning is needed beforehand for how to repay the debt[25].

6.4 Testing evaluation

This section will present a short evaluation of each of the testing approaches presented in the previous chapter. They will be evaluated by their benefits and drawbacks, based on what is found in the research. The findings are then examined by how it can contribute to the agile development. In quality, time and the learning curve if mentioned.

6.4.1 Unit testing framework

The unit testing framework presented in section 5.3 proposed a prototype of a framework, where the unit tests can use parameters and variables in their test cases. It is a relatively new method of testing and is known as ”parameterized unit test” (PUT). Moreover, it has been a part of the standard Java language test framework JUnit since version 4.

Even though the difference between a typical Unit test and PUT test are minimal. It introduces new approaches and possibilities for testing, in comparison to the conventional method of testing[42][41].

The original unit tests are a time-consuming process and sometimes introduces redundancy because the test is too coupled with a specific task and needs to be rewritten to suit a similar task[42]. As a result it would require more effort maintaining it for later changes in the development cycle.

The PUT gives the opportunity of implementing tests that are more generalized and can be reused for other tasks. This is because the values of variables are not tied directly to the test case itself. This enables for what is referred to as path exploration to increase code coverage[42][41]; It is the method of exploring different paths in the code by initiating variables so that they either pass or fail an if statement.

With effort, PUT, can be used in test case generation[41]. If PUT is designed as a general test, it can be adopted into a combinator. It would generate combinations of data for inputs, to achieve higher code coverage. It is a method, which has brought success in finding bugs in some projects[41].

In Agile this could be considered valuable, as it can reduce the time of writing tests. It provides a value to the customer by increasing the code coverage, easier to maintain and adapt

---

to the changes that are imposed in the later stages of the development process. However, it should be noted that creating generalized test cases requires more effort, but eventually, time is saved in regards to the possibilities.

6.4.2 Testing stubs framework

As pointed out earlier, agile aims to achieve a less time-consuming process when creating high-quality software. Moreover, the quality can be directly affected by how well the tests are implemented and enforced. However, most of the focus on testing is about the design and the functionality it can provide, and less about the test data.

Even though test data, is one of determining factors if a test will pass or fail. It is, however, less emphasized because manually creating extensive test data requires much effort. For that reason, the conventional practice is to create test data exercise partial components of the system. That said, this method can therefore not give a complete quality assurance of the whole system. Furthermore, generating test data manually introduces the human error, and therefore the tests can not asses the quality[39].

The framework introduced in the earlier chapter by Hassan, R and David Van Gilist (2010), is equipped with the possibility to auto-generate test data. This practice could significantly reduce the time and labor it otherwise would be required to generate test data, and at the same time increase code coverage[40].

However, as reminded by the author of the proposed framework, auto-generated responses is still in its infancy. Therefore, the capabilities to generate the response for more complex request requires further improvements.

6.4.3 Connectedness and exploratory testing

In the conventional way to go through with exploratory testing, it has a set of drawbacks in agile. It has been investigated by what it has for impact on the quality, labor, and maintainability. Moreover, it’s been found to potentially increase the technical debt, in several ways[35][36][37].

In the scenario that the team only rely on its intuition, and don not plan or document their approach for how to test each feature. It can cause a severe loss of time and resources because it increases the chances that the testers are overlapping each other and testing same thing multiple times[38]. The lack of explanation of how each of the tests was carried out, makes it harder to recreate and reevaluate the test for next sprint. Which essentially render the tests as useless and an increase of technical debt. Also, the lack of planning of what features to test and evaluate, can not generate a high test coverage.

Exploratory testing is the approach of having a tester to evaluate the feature, but having a process to help to guide the testers can make exploratory a beneficial approach.

It has also been shown that, the idea of having models to guide the exploration and observe the testers progress and approach. Have the potential to aid the agile team, by using it to initiate an exploring process and letting it create the tests that could be invoked by the tester[35][36].

This shows that there’s a potential for using the approach "connectedness” presented in the earlier chapter. As it can explore a REST service, by using the depth-first approach.
Moreover, it allows for planning as it needs a description of the diagram of relations between the resources, and resources can be excluded based on the testers intuition. The bottleneck, however, is that it adds extra complexity because of the need of knowing how to set up and define the diagram. Additionally, it is more adaptable to changes that might be imposed by the customer.

### 6.4.4 Model based testing

It was shown that MBT was found to have a great potential in the agile development[33]. As it is flexible, have a high test coverage and easier to maintain. The reason being that it is easier to adapt and change the properties in the model and generate new tests, compared to the conventional approach of updating every test manually.

Similar findings have also discovered that generating test can be beneficial for adding an extra variance to the test cases[34]. It gives the potential to avoid the cost that comes with a change of priorities or the specification[33][34]. This approach requires less effort to update and adapt the tests to fit the requirements compared to the conventional way of testing.

The primary requirement for tools to be used in agile, and in general, is for it to be easy to use and efficient at finding defects[34][33]. Moreover, the purpose of agile is to increase the value of a product, by not having processes and tools that are complex, and add extra labor.

A claim is that the MBT is considered to be simpler because it hides the complexity from the tester[33][34]. By using a universal language to describing the behavior of the test models at a more abstract level. The language consists of a set of action words, that is formed by the specific domain (area) [34]. This allows for others to create test models, without having in-depth experience of programming. Also, it could assist in drawing out the specification that is more close to the customer requirements, as they both speak the same language.

The drawback of the language is that it is limited to a specific domain, and needs to be tailored to suit other domains. Nevertheless, it solves the problem setting the boundaries for what is to be generated [34].

A significant problem with this approach is knowing if the generated tests fulfill the definition of ‘done.’ It is one of the critical requirement to unlock the full potential of MBT[33]. This requires that model has a substantial relation to the specification and an excellent continuous integration that can utilize the definition of done. Furthermore, it is also noted that it is hard to see the relation between the model and specification[34].

### 6.5 Conclusion

This chapter is aimed to evaluate what the benefits of the test methods and frameworks from the previous chapter. Then add the perspective of how it could assist in an agile development process.

The goal of agile is to produce software of high quality and bring the max value to the customer. While abiding by the constraint of cost, schedule, and scope. The following list summarizes what can be gained when applying the test methods and frameworks.

- **Unit testing framework:** It reduces the time to create and maintain tests. As it
provides a parameterized unit tests, which can be used to create fewer generalized tests. In turn, it makes it easier to maintain.

- **Stubs testing framework**: Auto-generating test data reduces the time and effort for creating test data. It gives a higher code cover, as it generates data in a way to covers more of the features. Furthermore, it assures the quality and the value of the features.

- **Connectedness**: It can reduce the time of exploratory testing because it can assist guiding the testers through the REST service.

- **Model-based-testing**: It assists in creating future value because it is flexible and easier to maintain. Furthermore, the abstract language lowers the knowledge barrier for third parties. Due to that, it generates value by allowing the customer and the tester speaking the same language, as it can derive the models closer to the requirements.

The Table 2, shows where the frameworks and test methods would be suitable for the continuous delivery development process. The performance column is blank, because none of the test methods or frameworks include the ability to stress test the performance of the system, as they are meant for functional testing.

<table>
<thead>
<tr>
<th>Tests</th>
<th>Integration tests</th>
<th>Acceptances tests</th>
<th>Performance tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit test framework</td>
<td>Compatibility testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stubs test framework</td>
<td>Replicate services and compati-bility testing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connectedness</td>
<td></td>
<td>Availability tests of resources</td>
<td></td>
</tr>
<tr>
<td>Model-Based testing</td>
<td></td>
<td>Behavior tests of the components</td>
<td></td>
</tr>
</tbody>
</table>
7 Conclusion

This thesis aimed to provide research about the testing of REST services, define requirements for test framework, and evaluate how it would benefit agile development. The thesis set out to find answers to the following questions defined in the problem statement:

- What research is there about methods and test framework, that is within functional testing, for RESTful web services.
- What functionalities should a test framework have. Based on the findings of the test methods and test framework from the previous question.
- How would the defined functionalities from the previous question, assist an agile team?

7.1 Research about testing REST

The research was conducted as literature review, based on the database provided in chapter 3 (method). The objective was to find test methods and test frameworks, in the area of functional testing, for RESTful web services. From our literature review, we discovered the following:

- **Unit testing framework:** The framework proposed by the authors Chakrabarti, S., and Kumpar P. (2009), is a black-box testing framework. It validates the responses, without concerning itself with the internal working of the system. Since the client needs to keep track of its state, the framework solve this by the ability to pass data between tests.

  As a result of REST not having constraints on what type of data can be transferred or received. The framework is equipped with plug-in validators and composite validation, it is to verify that type format is correct, and it is the expected type.

- **Stubs testing framework:** The second framework, proposed by Hassan, R and David Van Gilist (2010). Presents a framework used for white-box testing, which is meant to substitute REST service that is not available in the test environment. It is designed to behave similarly to how REST behaves to requests.

  It generates stubs, based on the interface specification. The stub have the capability to provide direct access and manipulate a resource, similar to as a REST service. The validation is done through the interface of the stub, which specifies what services are available and by what operations. Furthermore, it also validates the parameters of the URL, as there is no universal standard to what the parameters need to conform to.
The responses to the request, can either be of data-centered, which is a fixed data set of responses, or logical-centered, that generate a response in regards to the input.

- **Connectedness:** This is a method, which test if the service conforms to the HATEOS constraint. It enforces that all resources should be interlinked, for the client can access all the resources seamlessly. This method involves creating a binary tree representation of the service, for them to be searched through with depth-first algorithm. The number of resources available is then verified with the counted number of resources the method have visited, to confirm that all resources are reachable from the base resource.

- **Model-Based testing:** The method of model-based testing, is a research project to generate tests based on a model of the systems behavior. This is in order to generate tests so the behavior is extensively tested. It uses abstract language to describe the models which are then used for generating test cases.

### 7.2 Functionality for test frameworks

The second question of this thesis was to answer what functionality can be used to evaluate test framework for REST services. From the previous question, we gained knowledge of about testing techniques and frameworks.

We approached this by looking at what functionalities are relied upon for methods, and what functionalities are provided by the frameworks. From that, we could derive a list of functionalities and summarize it as following:

- **HTTP:** The test framework should support the following HTTP operations, GET, POST, PUT and DELETE.

- **General-purpose language:** The framework requires to able to express the similar operations that are found in the traditional programming language e.i if, else, and, or, variables, parameters, nested calls, lists, etc.

- **Auto-generated tests:** The framework should consider allowing for using abstract language, to create models which are later used for generating tests cases off.

- **Stubs:** The stubs should have the ability to define an interface, which behaves similarly to a REST service. There should be a two steps validation, first validation by the interface then the parameters. Before accessing the resources, the parameters should be validated by its order and types. Furthermore, auto-generated responses should also be included, as it helps to achieve a higher test coverage.

### 7.3 Agile aspect

The last question of the thesis is to answer in what way can the methods and frameworks found in the previous question, assist an agile team. We looked at the topics of the functionalities and conducted a literature review to get an understanding of the benefits and
drawbacks. The findings are then evaluated in regards to agile’s view on time, quality and learning curve. The following list describes what would be achieved with the functionalities defined in the previous question.

- **Time**: Time could be saved by the following functionalities: generalized parameterized unit tests, auto-generated tests, connectedness exploratory testing. However, creating generalized PUT and dynamic responses requires more effort than the conventional approaches.

- **Learning curve**: The model-based functionality offers the ability to represent the behavior of a system in an abstract language. It is shown to make it easier for third-parties to get involved without having in-depth knowledge.

- **Quality**: The functionality of auto-generated cases has the potential to be valuable, as it could provide numerous tests to complement manually created tests. It could achieve higher code coverage, and assure a better quality of the software.

  With a general-purpose language, it is possible to implement the connectedness method. It can be used for exploratory testing, and guide the team finding bugs and improve the quality of the service.

  Auto-generated test data is also useful when wanting to cover more of the features and assure the quality.
8 Summary and Future Work

8.1 Summary

The general aim of this thesis is to define a set of criteria that can be used for evaluating test frameworks for RESTful web services. The thesis aims to objectively achieve the following: provide research about methods and frameworks, define a set of functionalities for a framework and evaluate how it can contribute to agile development.

The first stage of this thesis, begins with examining the REST architecture, its constraints and concepts. The next stage, begin with presenting the test frameworks and test methods, it includes explanation how the they work, its purpose and underlying functionalities. The functionalities are then evaluated based on what it adds to the agile development.

The thesis discovered there were two types of methods and frameworks. Methods being connectedness and model-based testing, and the frameworks, a unit test framework and a stub framework. We found that test frameworks which have general-purposed language, auto-generated test cases and test data, are preferable when testing RESTful web services. Additionally, it can also be stated that these features have potential in agile development, as they could increase the quality and efficiency.

8.2 Future work

While there is no intention to provide any further contribution to this topic. It can be of interest to use the findings of the thesis, and apply it in practice. An suggestion would be to find actual test frameworks, which conforms to the criteria, and compare them to popular frameworks e.g, SoapUI\(^1\).

The approach could be the following: Setup test scenarios that were presented in this thesis, implement the test cases, and then finally verify which framework were able to implement with less effort.

Furthermore, there were plans to carry out a poll, to see if there were any particular functionality that was considered important for testing REST. However, it was considered too late, and therefore rejected.

\(^1\)https://www.soapui.org/
References


OMG (2017). OMG unified modeling language, superstructure version 2.5.1. [online] omg. Available at: https://www.omg.org/spec/UML/2.5.1/PDF


