UNIVERSITY OF MANNHEIM

Using Test Sheets for asynchronous testing of Real Time Software

Master Thesis

submitted: July 2016

by: Denys Zalisky
dzaliskyi@mail.uni-mannheim.de
born November 29th 1991
in Svetlovodsk

Student ID Number: 1440397

University of Mannheim
Chair of Software Engineering
D – 68159 Mannheim
Phone: +49 621-181-3912, Fax +49 621-181-3909
Internet: `http://swt.informatik.uni-mannheim.de`

Abstract

- 2-3 sentences - current state of art
- 1-2 sentences - contribution to improvement
- 1-2 sentences - specific result of the paper and main idea behind it
- 1 sentences - how result is demonstrated and defend

While providing of simple way for test description is a hot topic in software development. There is no software developed for a realization of Test Sheet concept, pragmatic way of defining tests which lays between two extreme paradigms FIT and hard coded test definitions.

This paper describes processes of design and implementation of the Test Sheets' concept together with integration of the product to business processes of figo GmbH for a real-time testing/validation of internet banking web pages.

Result of this research is following: developed conventions for Test Sheets definitions in particular use case, implemented interpreter from Test Sheets to executable JavaScript code.

Conventions and the code listing of main module together with example of executable JavaScript file are provided as well as statistics regarding improvement of user experience and overall system fault prevention improvements.

Some feedback from Bianca and Sebastian + statistics regarding user experience improvement

Contents

List of Figures

List of Tables

1 Introduction

- What precisely did I answer

    what question did I answer

    why should the reader care

    what larger question does this address

- What is my result

    What new knowledge have I contributed that reader can use else where

    What previous work do I build on

    What precisely and in detail my new result

- Why should the reader believe in my result

    What standard was used to evaluate the claim

    What concrete evidence shows that me result satisfies my claim

Relevance of the topic and the necessity for scientific investigation: No researches found regarding semi automated tests generation for web page verification.
Practical and theoretical value of the topic: Implementation of enginee for Test Sheets with application of software design and development practices.
Motives for choosing a particular topic: Necessity of tests defined by non-developers for figo for a real-time testing (will be provided later)
Research problem and why it is worthwhile studying - definition of convention for test sheets definition, usage of test sheets for testing of asynchromous systems in a real-time.
Research objectives - design and development of software for translation of test sheets in to executable java script for testing asyncronous calls to external system.

Structure of the thesis : A paragraph indicating the main Contribution of each chapter and how do they relate to the main body of the study Limitations of the

2 figo GmbH

The German banking system is divided by three large sectors: private, public, and cooperative. The cooperative sector is represented by 1,144 credit unions and 2 cooperative central banks. The public sector employs 431 savings bank, 10 land banks and other institutions. Private banks represented 4 transnational banks, 42 investment banks, and 176 regional and other banks. There is also operating are 167 registered branches of foreign banks, including 60 investment banks[5].

The introduction of Payment Services Directive (PSD) and PSD2 by European Commission in EU together with initiatives of UK Government regarding API provision and standardization have obligated banks with the implementation of on-line access points to their services[22][24][4]. Within the Single European Payment Area acceptance of directive by European Bank Authority scheduled within 2017 year[8].

figo GmbH is a financial technology (FinTech) company with the headquarter located Hamburg, Germany. It was founded in 2012 with the mission "to build the backbone of next generation financial services"[16]. Currently, the API is fully functional in Germany, partly in Austria and England[10][11].

figo Connect API was created with the aim to accelerate innovations in the FinTech area and to allow figo's partners to offer products with real added value[29]. It enables developers, startups and even banks to connect to every financial service. These partners can access every bank account (current, savings, loan, securities, etc), credit card, eWallet and other financial services l(i.e. PayPal) through one single REST-API [29][16][32]. The list of figo's partners and customers together with there use cases can be found via following link: `http://figo.io/use_cases.html`.

2.1 IT infrastructure. Banking Server

The high level IT infrastructure of figo GmbH consists two parts (Fig. 2.1). The **API Server** implements interfaces to figo's customers and partners for accessing banking information and services (lays outside of the paper's scope). The **Banking Server** implements connection to banks via three possible communication channels, their description provided below together with basic motivation for each of them.

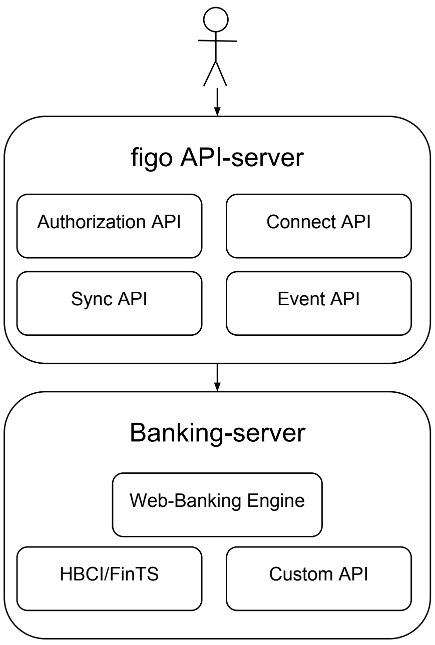


Figure 2.1: figo GmbH high level architecture

2.1.1 Banking Server Architecture

Banking server has three parts for communication with banks via three separate channels. Each of them is a realization of different technology in a same programming language - javascript.

Custom-API is responsible for connection to custom APIs provided by banks. It implements the client for custom banks APIs. Some of them provide full functionality while some only partial. All this APIs vary in their structure and functionality but most of them are an implementation of REST-API specification.

HBCI+/FinTS is responsible for connection to banks' interfaces via Home Banking Computer Interface (HBCI). It is an implementation of an *adapter OOP pattern* for jsHBCI library. HBCI is an open publicly available protocol. Its specification was originally designed by the two German banking groups *Sparkasse and Volksbanken und Raiffeisenbanken* and *German higher-level associations as the Bundesverband deutscher Banken e.V.*. [12]

Web-Banking Engine is responsible for communication with banks which does not provide API or HBCI. This is an implementation of a *factory OOP pattern* for scraping libraries. Here figo GmbH uses *web scraping* technology to perform interaction with internet-banking web pages. From the banks perspective the interaction looks completely like direct communication with an user, while an user does not feel the difference between interaction via Custom-API or HBCI or Web-Banking Engine while accessing his bank or service via figo API. This is the most sensitive part from the developer's perspective since every change to the bank's web page can leads to failure of the specific scripts.

2.2 Requirements

Company's **business requirement** were a realization of mechanism for detection of changes in banks web-pages. The tool should provide business staff of the company to define business scenarios in a non-technical way. After definition such scenarios must be executed over web-page. Create, Read, Update, Delete (CRUD) operations for this scenarios must be performed with minimal involvement of software developers.

Technical requirements introduced by figo GmbH were following: First, test execution must be performed in a timely fashion for an opportunity to detect changes as early as possible to prevent attempts of customers' communication with not available banks or services. Consequently tests execution have to be fast. Next software developers must be notified about script failure as soon as it was detected to improve response time. Next, scraping scripts are communicating with web-pages in a real-time and asynchronous fashion. Further, the language of implementation must be javascript for low entry level of developers in a future maintenance process. Last but not least, due to the nature of data represented on a web page the comparison between actual and expected results must have two possible options: object keys comparison and object key-value comparison.

3 Testing. Existing approaches. Test Sheets

3.1 Software testing

Software testing is an important technique to evaluate and assess software's quality and reliability, it provides possibility to determine whether the development of product conform the requirements. The variety of tests coincides the variety of requirements for the software under test. At least 30% of the price of software project is the cost of software testing [37]. A General requirements to test definitions are following: fast, independent, repeatable, self-validating, timely. More detailed information regarding requirements can be found via following URL: `http://www.extremeprogramming.org/rules/testfirst.html`

The economical costs of unexpected software behavior (bug, failure or error) can go up to several millions or even billions of USD.
Web application failures only in USA lead to looses of \$6.5 million per hour in financial services and \$2.4 million per hour in credit card sales applications[36].
Alarm-management fault was one of the reasons of the black-out occurred in the northeastern US on August 2003. Estimated costs were between US\$7 and \$10 billion[25].
In general, according to report made by US National Institute of Standards and Technology (NIST) in 2009 the estimated economy loses were \$60 billion anually as associated with developing and distributing software patches and reinstalling systems that have been involved, together with losses in productivity due to errors and malware infections[25].

3.2 Existing approaches for test definitions

In most of the cases developers are using testing frameworks or internal DSLs which requires usage of formal programming languages. This in a result makes writing of tests not different from the programming. It requires knowledge of languages and understanding of basic programming concepts from everyone who is involved in the creation of a test, reading its result or updating/deleting the test. Since it is common practice to base tests on top of business requirements tests should be used by different shareholders who define this requirements. Below provided brief overview of different testing approaches and their analysis with respect to modern business requirements.

3.2.1 xUnit

xUnit is a family of unit testing frameworks with shared architecture and functionality which is derived from Smalltalk's SUnit, designed for tests automation[44][34]. The general simplicity and lightweight made them popular tool for Test Driven Development[34].

xUnit basic features implemented by all members of the family provide functionality to perform following tasks[44]:

1. Specify a test as a *Test Method*;
2. Specify the expected results within the test method in the form of calls to *Assertion Methods*;
3. Aggregate the tests into test suites that can be run as a single operation;
4. Run one or more tests to get a report on the results of the test run

Test ***cases*** in xUnit are defined as a methods united in to ***suites*** with shared preconditions called ***fixtures***. Cases or suites are executed by a ***runner*** which compares actual and expected results using ***assertion*** function.

The family includes a wide variety of implementations for multiple programming languages, and diverse enhancement of functionality (e. g. code coverage statistics, assertion add-ons etc.)[35].

Definition of tests with formal general programming language from one side makes tests definition and execution fast but in a same time makes impossible the creation of tests for people without developer background.

3.2.2 Fit

Framework for Integrated Test (Fit) - is a way of defining test cases with HTML pages. It enhances the communication and collaboration connecting customers and programmers. Moreover, it creates feedback loop between them. Fit automatically checks HTML pages against actual program[13].

Fit reads tables in HTML files, each table is interpreted by a *fixture* written by programmers. This fixture checks the examples in the table by running the actual program[14].

Programmers use a *ColumnFixture* to map the columns in the table to variables and methods in the fixture. The first columns provide correspond to variables in the fixture. The last column has the expected result and corresponds to the method in the fixture[14].

Different implementation provide different user experience from Wiki pages (i.e. FitNess) to complete standalone application with internal functionality for tables definitions and tests execution (i.e. GuiRunner).

With Fit, customers can provide more guidance in a development process by lending their subject matter expertise and imagination to the effort[13] requiring developer to write only *fixture*, the middleware between tests and code. In the same time it provides media between CRUD operations over tests and the results of their execution for people without development experience.

3.2.3 Cucumber

Cucumber is a Behavior Driven Development tool which allows users to define tests specifications with *Gherkin*, the language which is structured plain text with support of internationalization for 60+ different languages[7].

Feature files written in Gherkin consists of plain text and general key-words are used for identification of following concepts:

- Feature under the test: *Feature*;

- Test scenario: *Scenario*;
- Scenario Outline: *Scenario Outline*
- Test Steps: *Given, When, Then, And, But*;
- Test Background: *Background*;
- Test Examples: *Examples*

Specific characters are used for identification of:

- Step Inputs: String - *"""*, Table - |;
- Step Tags: @;
- Comments: #

Step definitions are written by developers. Definition parses feature file with attached pattern to link all the step matches and code executed by cucumber when match is met.

Human readable input and output of the tests defined in gherkin makes cucumber a good media for communication between people who create requirements and those who are trying to meet them. But in a same time software development experience required for writing code with step definitions.

3.3 Test Sheets

Test Sheets is a representation for tests a developed with the goal to combine the power and completeness of formal programming language with a representation that is easy to understand and work with even for people with little IT knowledge[26].

Test Sheets approach uses usual spreadsheet for definition of test and representation of their result.

- Rows - represent operations being executed;
- Columns - variables for input or output parameters;

The actual content of a cell can be made dependent on other cells by addressing them via their location. Just like in Fit result of tests execution is provided in a same table with coloring and actual return are separated by "\" symbol in case of failing test[27].

There are three types of test sheets which can be used in combination:

- **Basic** - order of test step execution is defined by order of rows in a table;
- **Non-Linear** - order of test steps execution is defined by finite state machine with states represented by test steps and transition function by step execution results or number of test step execution;
- **High-Order/Parametrized** - The actual value used for Parameterized Test Sheets is specified by a Higher-Order Test Sheet in a column with letter Parametrized Test Sheet refers to.

The main benefit of this approach over *xUnit*, *Cucumber*, *Fit* is an exclusion of software developers from CRUD test operations which can be performed by user without prior development experience with usage of any spreadsheet editor.

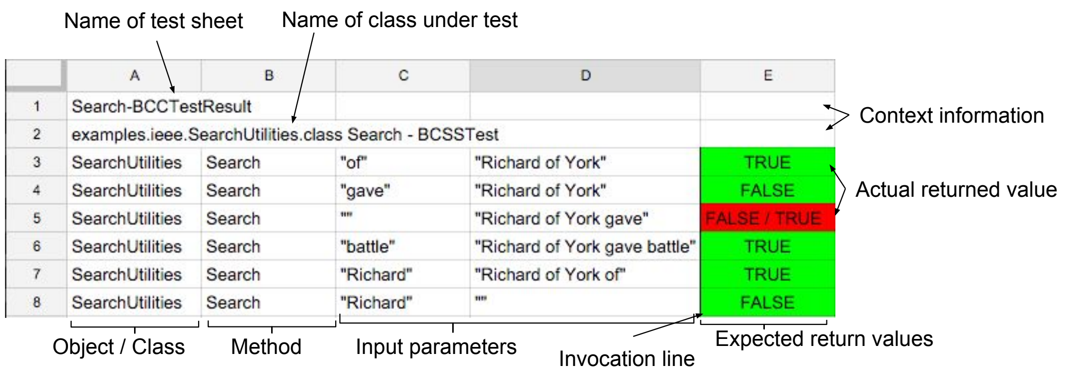
3.3.1 Basic Test Sheets

A Basic Test Sheet is a type of Test Sheets with sequential tests execution without parametrized values, but with possibility to use references between cells for definition of values. The rows content is structured as following:

- *Test name* - first row;
- *Class/Module under test* - second row;
- *Test step* - all next rows represent method calls;

Values in columns cells of which belong to the test step has net purposes:

- *Object Under Test* - cells in a first column;
- *Method Under Test* - cells in a second column;
- *Input Parameters* - cells in a next column before invocation line;
- *Invocation Line* - cells in a delimiter column between cell(s) with biggest number of input and column with expected return value;
- *Expected Return* - cells in a column after invocation line;



	A	B	C	D	E
1	Search-BCCTestResult				
2	examples.ieee.SearchUtilities.class Search - BCSSTest				
3	SearchUtilities	Search	"of"	"Richard of York"	TRUE
4	SearchUtilities	Search	"gave"	"Richard of York"	FALSE
5	SearchUtilities	Search	""	"Richard of York gave"	FALSE / TRUE
6	SearchUtilities	Search	"battle"	"Richard of York gave battle"	TRUE
7	SearchUtilities	Search	"Richard"	"Richard of York of"	TRUE
8	SearchUtilities	Search	"Richard"	""	FALSE

Figure 3.1: Basic Test Sheet

The description provided above is shown in a summarized example from the web site of Chair of Software Engineering of University of Mannheim (Pic.: 3.3.1).

The software developed within the scope of this research is a proof of concept for an usage of Test Sheets for scenario testing of asynchronous real-time software. The scope of this paper covers only Basic Test Sheets. For more details about Test Sheets and Research Topics of the Chair of Software Engineering please visit `http://swt.informatik.uni-mannheim.de/de/home/`.

4 Real-Time Software and its Testing. Web scrapping

Donald Gillies defined a real-time software as a system: *"[...] in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred."* While Robert L. Glass[42] defines this term as: *"[...] a software that drives a computer which interacts with functioning external devices or objects. It is called real-time because the software actions control activities that are occurring in an ongoing process".* Within this research we will define *Real-Time software* as a combination of this to definitions which covers both logical and time correctness as well as an interaction with external systems controlled by the ongoing process.

The Web-Banking engine of figo GmbH matches this definition due to the following facts. First, it performs communication with external systems (Web Banking HTML pages). Next, its result correctness depends time restrictions (if child process responsible for script execution was not finished within 1200 seconds, and with each task performed within 0.5 seconds it is treated as failed) as well as logical correctness depended upon ability of the script to perform necessary actions for a fulfillment of a requested task.

Tsai, Fang and Bi[46] state that testing and debugging of real-time software are very difficult because of timing constraints and non-deterministic execution behavior. In a real-time system, the processes receive inputs from real world processes as a result of asynchronous interrupts and it is almost impossible to precisely predict the exact program execution points at which the inputs will be supplied to the system. Consequently, the system may not exhibit the same behavior upon repeated execution of the program. In addition, in a real-time system, the pace of an execution of processes is determined not only by internal criteria, but also by real world processes and their timing constraints.

The general testing and debugging strategy (Fig. 4) described by Tsai, Fang and Bi [46] consists of four steps. At the *first* step, a set of events which can be used to represent the program's execution behavior at a specific abstraction level and the event key values which describe these events are identified. At the *second* step, the execution data, containing the event key values identified in the fist step, is collected using the real-time monitoring system. The first and second steps constitute the monitoring phase. After the data is collected, at the *third* step, the collected data is processed in an off-line mode to construct logical views to represent the program's execution behavior. At the *fourth* step, analysis algorithms will be employed to identify fault units and to report to the user. Finally, the user can use this report to decide next monitoring and debugging cycle for a lower level abstraction. In this paper the provided strategy is applied for each individual test.

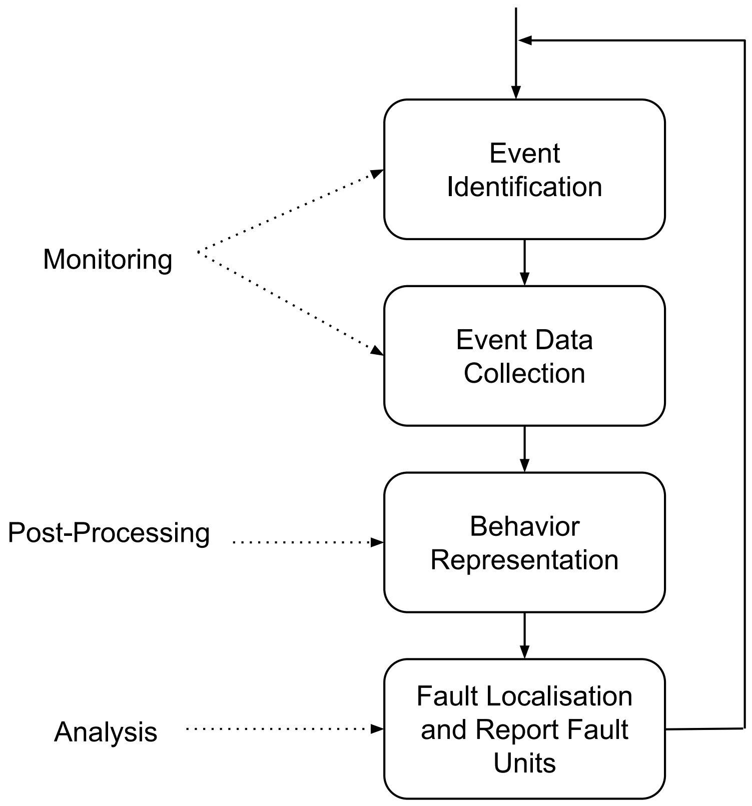


Figure 4.1: General Testing and Debugging Strategy[46]

4.1 Web scraping. CasperJS

Web scraping is a method for extracting information from web pages[30]. This technique is useful when you want to do real-time, price and product comparisons, archive web pages, or acquire data sets that you want to evaluate or filter[3].

Web scraping works via interaction with Document Object Model (DOM) of the web page it is an useful technique for real-time data extraction from human readable web-pages. W3Council defines DOM[28] as "... a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents."

CasperJS is an open source navigation scripting & testing utility written in javascript for the headless browsing. It eases the process of defining a full navigation scenario and provides useful high-level functions, methods & syntactic sugar for doing common tasks for interaction with DOM of the web page[6]:

- defining & ordering browsing navigation steps
- filling & submitting forms
- clicking & following links
- capturing screenshots of a page (or part of it)
- testing remote DOM
- logging events
- downloading resources, including binary ones
- writing functional test suites, saving results as JUnit XML
- scraping Web contents

Usage of this tool gives an ability for figo GmbH to pragmatically imitate user behavior on a web page. Which includes filling login forms, filing forms for retrieving service information and perform other business activity available via a service web site. CasperJS instantiates web-page or its part defined as tree of DOM selectors. Page content includes both HTML and javascript/jQuery code from the web-page addressed with provided URL. Instantiated page or its part is stored in memory and can be accessed for further manipulation by CasperJS methods as any other object instance.

5 Asynchronous programming. Strategies and performance

5.1 Asynchronous programming

Asynchronous programming is the programming model in which operations should be done are interleaved with one another within the single control thread. Analogy to it can be package multiplexing in a Computer Networks.

To compare to multi-threaded systems are synchronous and allow execution of one task per unit of time blocking execution of other tasks untill programmer will perform explicit control delegation to other task.

Generally asynchronous systems are easier to control and to develop rather then multi-threaded, and they perform better then synchronous in following cases[18]:

- Large number of tasks and at least one task likely to make progress;
- A lot of I/O operations;
- Tasks are independent from one another;

5.1.1 Node.js

Node.js is an asynchronous event driven framework designed to build scalable network applications. Node.js is similar in design to and influenced by systems like Ruby's Event Machine or Python's Twisted with difference that it presents the event loop as a language construct instead of as a library[1].

Node.js applications are written in javascript - a lightweight dynamic scripting multi-paradigm language with first-class functions and prototype-based inheritance which provides both object oriented and functional programming approaches.

Prototype. When it comes to inheritance, JavaScript only has one construct: objects[17]. Also called classless, prototype-oriented, or instance-based programming[20]. Each object has an internal link to another object called its prototype. That prototype object has a prototype of its own, and so on until an object is reached with **null** as its prototype. **null**, by definition, has no prototype, and acts as the final link in this prototype chain. JavaScript objects are dynamic "bags" of properties (referred to as own properties). javascript objects have a link to a prototype object. When trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototype, and so on until either a property with a matching name is found or the end of the prototype chain is reached[17] This type of inheritance provide objects inherit both instance and class methods and properties.

This framework has powerful ecosystem - npm which includes *npm* - package manager for node.js, *npm Registry* - public collection of packages of open-source code, *npm comand line clinet* which allows developers to install and publish those packages. On a diagram (5.1.1) stated the comparison of npm with other package managers made by module counts, for more up to date information please visit `http://www.modulecounts.com/`

In the same time there is a downside of such ecosystem. The case of *left-pad* the module which was deleted from the npm due to copyright concerns caused to failure during the build process for thousands of projects[39]. Further, there are three factors explained by Sam Saccone [45] [40] which make npm a root source of hackers attack. The first reason is caused by usage of SemVer for version controlling which does not lock dependencies to a specific version [45]. The second reason is the lack of automatic npm user log-out. Since npm can run arbitrary scripts on install that means that any user who is currently logged in and types npm install is allowing any module to execute arbitrary publish commands [45]. The last reason is dependent on that fact that a singular npm registry is used by the the large majority of the node.js ecosystem on a daily basis [45]. The high-level architecture of node.js looks following:(5.1.1).

Node core is a javascript library (called node-core) that implements the high-level node.js API.

Bindings responsible for wrapping and exposing *libuv* and other low-level functionality to javascript.[38]

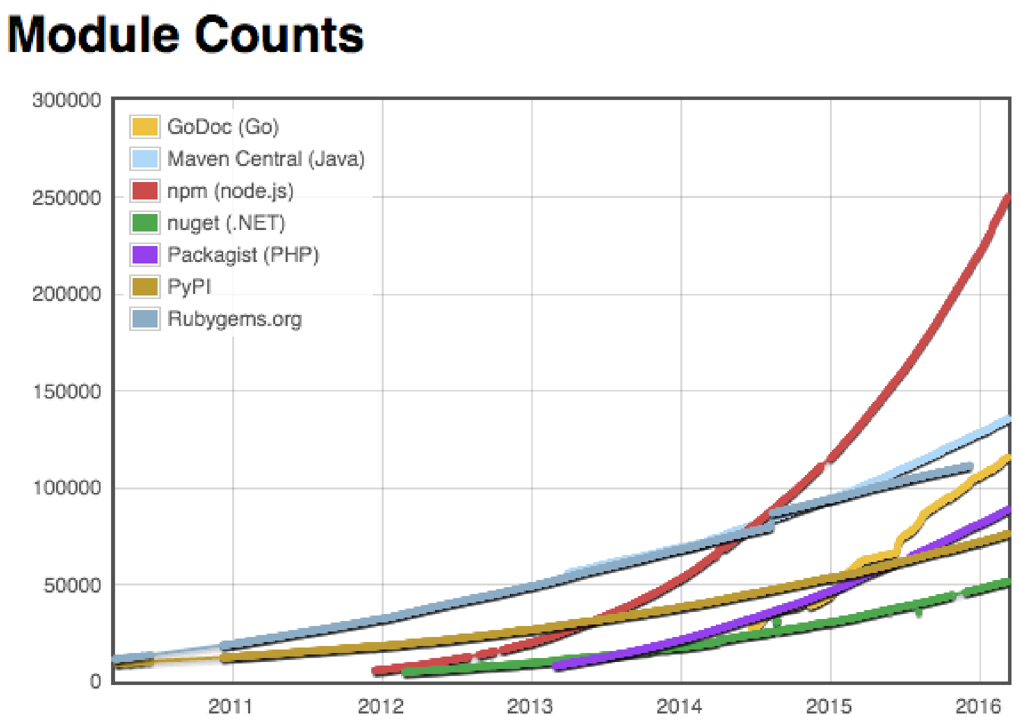


Figure 5.1: npm comparison with other package managers[23]

Non blocking I/O provided by libuv[1][38]. Which is "a multi-platform support library with a focus on asynchronous I/O. "[31] with following properties[19]:

- Abstract operations, not events
- Support different nonblocking I/O models
- Focus on extendability and performance

V8/Chakra the JavaScript engine originally developed by Google for the Chrome browser/ Microsoft for IE 9 browser"[38]

5.1.2 Event handling

Event is a core concept of asynchronous programming in node.js. All objects that emit events allows one or more functions to be attached to named events emitted by the object. When event is emitted all of the functions attached to that specific event are called synchronously[9]. Node.js event handler is an implementation of *reactor pattern*. The illustration of process life-cycle is shown on Fig. 5.1.2:

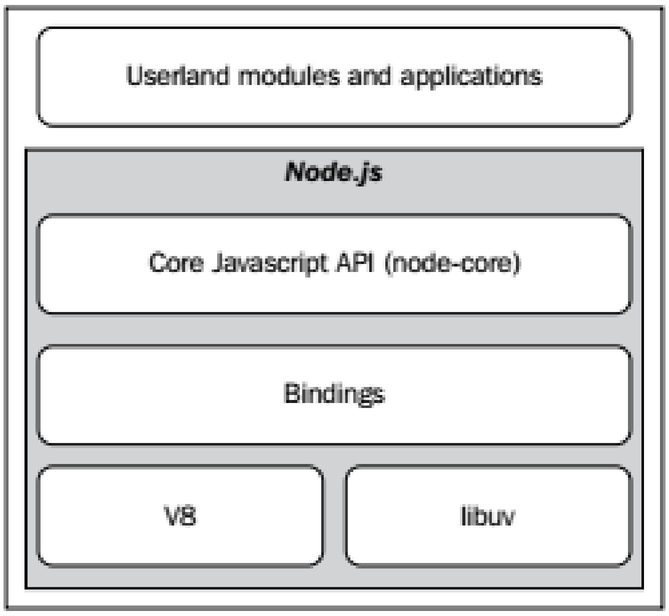

Userland modules and applications
Node.js
Core Javascript API (node-core)
Bindings
V8
libuv

Figure 5.2: Node.js architecture [38]

1. The application generates a new I/O operation by submitting a request to the *Event Demultiplexer*. The application also specifies a *listener*, which will be invoked when the operation completes. Submitting a new request to the Event Demultiplexer is a non-blocking call and it immediately returns the control back to the application.

2. When a set of I/O operations completes, the Event Demultiplexer pushes the new events into the *Event Queue*.

3. At this point, the *Event Loop* iterates over the items of the Event Queue.

4. For each event, the associated listener is invoked.

5. The listenr, which is part of the application code, will give back the control to the Event Loop when its execution completes. However, new asynchronous operations might be requested during the execution of the listener, causing new operations to be inserted in the Event Demultiplexer, before the control is given back to the Event Loop.

6. When all the items in the Event Queue are processed, the loop will block again on the Event Demultiplexer which will then trigger another cycle.

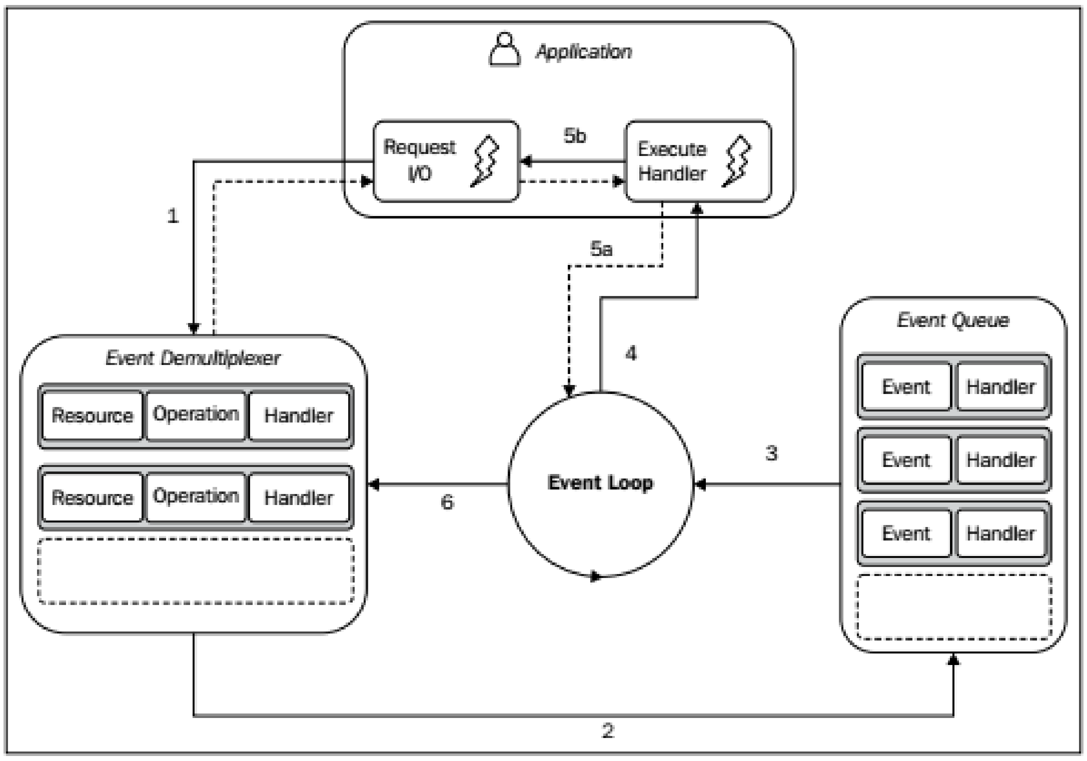


Figure 5.3: Node.js event handling system [38]

5.2 Asynchronous handling strategies

5.2.1 Continuation passing

Presence of functions as a first class citizens in javascript allows direct usage of functions for handling asynchronous program behavior. *Callbacks* are handlers for the reactor pattern described before. They are similar to *visitor pattern* in OOP, they also represent an operation to be preformed on the elements of an object structure and they let define a new operation without introducing any changes to the definition of the object.

Callbacks implement continuation-passing-style from FP. By convention callbacks must be passed as the last argument and accept two parameters, the first is an error and the second is a data to be processed further.

There are two main negative sides of using callbacks. One of them is so called *callback hell* occurs due to the abundance of closures and in-place callback definitions. This makes the code hard to be read because of high level nesting, as well as written due to a scope of nesting and difficult to manage because of possible memory leaks created by closures. Another negative side is called *releasing Zalgo*[15], it occurs only in case of inconsistent function behavior, when under some hidden conditions a function performs synchronous action but some under other - asynchronous.

5.2.2 Imperative

Before starting with imperative strategies for handling of asynchronous data flow we would like to show the *concept of duality* (Fig. 5.2.2). showed by Erik Meijer `https://channel9.msdn.com/Events/Lang-NEXT/Lang-NEXT-2014/Keynote-Duality` and hardly used by Kris Koval in his *General Theory of Reactivity* `https://github.com/kriskowal/gtor/`. This chapter is based on works of Kris Koval, Erik Meijer, Conal Eliot and Mark S. Miller in a field of Reactive Programming.

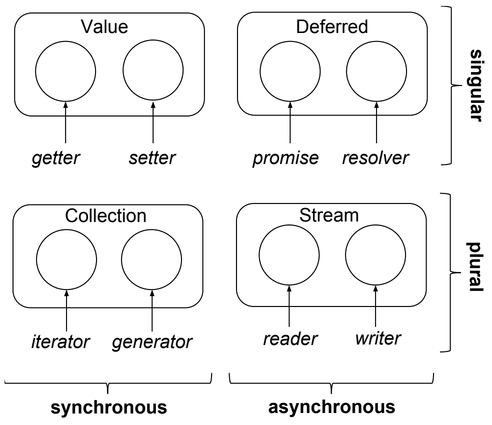


Figure 5.4: Duality Matrix [21]

The explanation of asynchronous patterns in this chapter will be performed via mapping of *synchronous* patterns to the *asynchronous*. Another dimension of mapping is *singular* vs *plural*.

During communication can occur the problem caused by the fact that different parts for the dialog can have different load and different performance. The first situation is **Fast producer - slow consumer** - get of one entity works faster then set of next entity in the chain. This situation occurs when values are ***pushed*** by producer. The second situation is **Slow producer - fast consumer** - get of one entity works slower then set of next entity in the chain. This situation occurs when values are ***pulled*** by consumer. Solution of this problems lays on scope of the system design which should allocate push and pull entities in a appropriate sides of the communication channels.

Synchronous: **Value** is a singular unit data. Its duals are *getter (pull)* and *setter (push)*. Setter accepts value to be assigned and return nothing and getter accepts nothing and returns a value. The chaining process can be performed here by applying setter to getter and getter to setter and by applying same logic to their analogs for further entities. **Collection** is a plural form of the value. The duals of a collection are *iterator (pull)* and *generator (push)*. Iterator as a plural analog of getter, it accepts nothing and returns the element from collection. Generator is a plural analog of setter it accepts element to be added to collection and returns nothing.

Asynchronous: **Deferred** is analog of the value. The duals for it are *resolver (push)* and *promise (pull)*. The resolver is an asynchronous analog of setter. It accepts value which will be assigned as soon as it will be resolved. The promise is an asynchronous analog of the getter . It allows to obtain the value of the promise as soon as it will be resolved. The deferred concept guaranties unidirectional data flow which means that data can go only from resolver to promise. Further deferred entities guaranties asynchronism of execution for an operation which means they are "Zalgo safe"[2]. **Stream** is an analog of the collection. It can be treated as a collection of deferred elements. The duals for stream are *write (pull)* and *read (push)*. The read is an analog of iterator it accepts nothing and takes values from the stream. The write is an analog of generator it accepts values from the stream and returns nothing. As a plural analog of deferred it guaranties unidirectional data flow. The special case for streams in node.js are transform and duplex (transform) streams which are combination of read and write.

5.3 Performance

Following measures are taken from the article "Analysis of generators and other async patterns in node" by Gorgi Kosev (`https://spion.github.io/posts/analysis-generators-and-other-async-patterns-node.html`). There were no hardware characteristics provided for the experiment execution environment except following description[33]: "On my machine redis can be queried about 40 000 times per second; node's 'hello world' http server can serve up to 10 000 requests per second; postgresql's pgbench can do 300 mixed or 15 000 select transactions per second."

The performance metrics were taken from the experiment under the the conditions where all external methods are mocked using setTimeout 10ms to simulate waiting for I/O with 1000 parallel requests (i.e. 100K IO / s) [33]

pattern	**time(ms)**	**memory (MB)**
promises-bluebird	512	57,45
promises-bluebird-generator	364	41,89
callbacks	316	34.97

Table 5.1: Performance comparison of patterns for asynchronous information flow [33][2]

"The original and flattened solutions are the fastest, as they use vanilla callbacks" [2]

Note that this table has only fastest among promise objects and since there were no measurements performed for streams but according to their nature the most optimistic performance for them should be equal to the promise generator provided by Bluebrid `http://bluebirdjs.com/docs/getting-started.html`.

6 Architecture

The system consist of three piped object streams 6. Streams piping implements the idea of pipe '—' in Unix systems invented by Douglas Mcllroy. It enables the output of one program to be connected to the input of the next program. *Object stream (Stream in a object mode)* is a stream in which data treated as a sequence of discrete javascript objects. Further, piping of object streams allows to perform parallel executions which can be beneficial from the performance perspective.

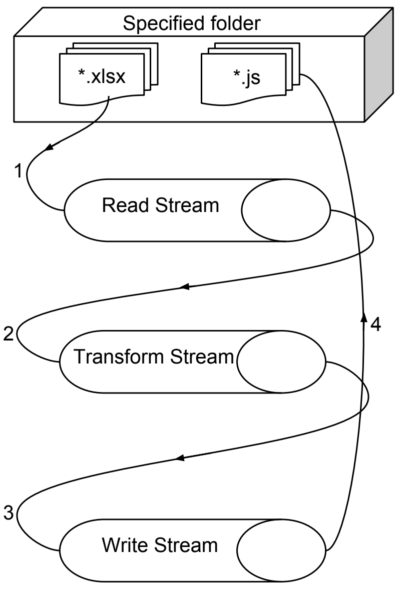


Figure 6.1: Information flow

1. ***Read Stream*** accepts directory address and *pulls* content of all .xlsx files together with their metadata from this directory including nested directories;
2. ***Transform Stream*** accept data object with file name, content, metadata from upstream, creates TestSheet schema and generates content of .js file implementing *interpreter pattern*;

3. ***Write Stream*** *pulls* data from upstream perform attempt to read representative .js file from specified folder if file exists and its last update date is older then for .xlsx file the next step will be skipped;
4. ***Write Stream*** if file does not exist or its last update date is earlier then last update date of .xlsx file it creates/overwrites .js file.

Pipe method of streams provide developers with opportunity to chain streams implementing different piping patterns:

1. *Combining* - encapsulation of sequentualy connected streams in to single looking stream with single I/O points and single error handling mechanism by pipeing readable stream in to writable stream;
2. *Forking/Merging* - piping single readable in to multiple writable streams / piping multiple readable streams in to single writable stream;
3. *Multiplexing/Demultiplexing* - forking and merging pattern which provides shared communication channel for entities from different streams, analogy can be computer networks.

As was already described streams are deferred analog of arrays, which allows to perform such operations as mapping, reducing and filtering. The process of transformation of xlsx files to js files in this application is treated as mapping process in general case, and filtering for avoiding of redundant file's overwriting.

The folder/file structure of the application looks as following:

- index.js
- package.json
- ReadMe.md
- lib/
  - scheme/
    - index.js
    - execution_scheme.js
    - order.js
    - scheme.js

  - stream/
    - index.js
    - read_stream.js
    - transform_stream.js
    - write_stream.js
  - template/
    - index.js
- test/
  - read_stream.js
  - write_stream.js
  - transform_stream.js
  - scheme.js
  - execution_scheme.js
  - order.js
  - template.js
  - doublers/
    - TestSheetOjbect.js
    - TestSheet.xlsx
    - TestSheet.js
- node_modules/

Creation of folders for scheme and template directories is made for purpose of expansion in case of adding Non-linear and/or HigherOrder Test Sheets. The entry point of the system ./index.js looks as following (Listing: 6.1):

```
var stream = require('./lib/stream');

stream.read(process.argv[2]).pipe(stream.transform).pipe(stream.
    write);
```

Listing 6.1: index.js

7 Design and Implementation

The implementation of each part of the system was made with application of Test Driven Development which guarantees full test coverage for the code. Used testing framework is *mocha* `https://mochajs.org/`. For creation of doublers (mocks, stubs and spies) for object and method was used *sinon.js* `http://sinonjs.org/`. Robet Cecil Martin[43] defined symptoms of not agile design:

- Rigidity - The design is difficult to change;
- Fragility - The design is easy to break;
- Immobility - The design is difficult to reuse;
- Viscosity - It is difficult to do the right thing;
- Needless complexity - Overdesign;
- Needless repetition - Mouse abuse;
- Opacity - Disorganized expression;

To avoid creation of such software John Dooley [41] and Rober Cecil Maritn [43] defined the principles of agile architecture.

- Closing - Encapsulate things in your design that are likely to change.
- Code to an Interface - rather then to an implementation.
- Do not repeat yourself - Avoid duplicate code.
- The Single-Responsibility Principle - A class should have only one reason to change from the business perspective;
- The Open/Closed Principle - Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification
- The Liskov Substitution Principle - Subtypes must be substitutable for their base types.
- The Dependency-Inversion Principle -

  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend upon details. Details should depend upon abstractions.

- The Interface Segregation Principle - Clients should not be forced to depend on methods they do not use.
- Principles of Least Knowledge - Talk to your immediate friends.
- Principle of Loose Coupling - object that interact should be loosely coupled with well-defined intefaces.

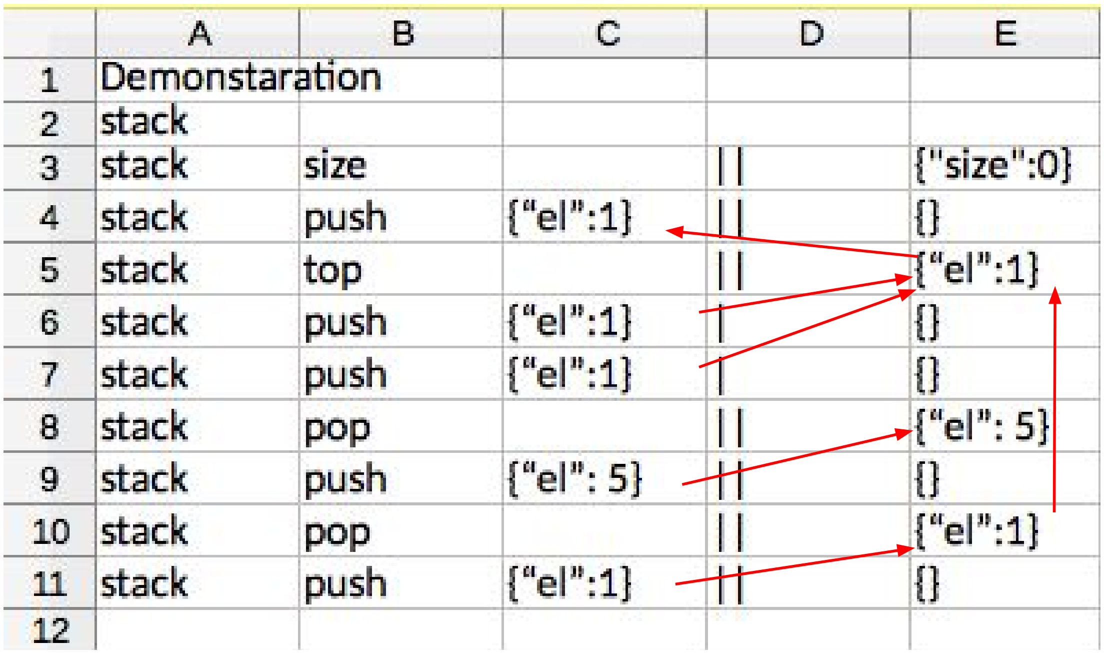
This principles have a suggestional nature and can not be strictly followed due to their mutually exclusive nature. Correspondence to each of them will be provided in the end of each section within this chapter.

For description of data structures processed by the system we will use following module (Listing: 7.12) which implements stack with asynchronous methods (respond time for each method is 10 milliseconds). Usage of *setTimeout()* method guaranties its asynchronism while the fact that timeframe for response hardcoded into scraping scripts guarantees that this method is valid for usage as a proof of concept in current research topic.

```
var stack = [];

module.exports = {
    push: function(el, cb){
        return setTimeout(function(){
            stack.push(el);
            return cb(null, {});
        }, 10);
    },

    pop: function(cb){
        return setTimeout(function(){
            return cb(null, stack.splice(-1)[0]);
        }, 10);
    },

    top: function(cb){
        return setTimeout(function(){
            return cb(null, stack.slice(−1)[0]);
        }, 10);
    },

    size: function(cb){
        return setTimeout(function(){
            return cb(null, {size: stack.length});
        }, 10);
    },
}
```

Listing 7.1: stack.js

The Test Sheet for test coverage of this module looks as following (Figure: 7.1). Note that this test coverage was made with purpose to show handling of asynchronous testing of real-time software but not to cover stack module. There are both types of comparison in invocation line **D**. Red arrows indicates references withing this test sheet.



	A	B	C	D	E
1	Demonstaration				
2	stack				
3	stack	size		| |	{"size":0}
4	stack	push	{"el":1}	| |	{}
5	stack	top		| |	{"el":1}
6	stack	push	{"el":1}	|	{}
7	stack	push	{"el":1}	|	{}
8	stack	pop		| |	{"el": 5}
9	stack	push	{"el": 5}	| |	{}
10	stack	pop		| |	{"el":1}
11	stack	push	{"el":1}	| |	{}
12					

Figure 7.1: Test Sheet coverage for stack.js

7.1 Read Stream

This part of the system is an implementation of combining streams pattern. From the internal view it consists of two streams. First stream performs recursive search for files in a provided directory and returns array with absolute paths to them. Second stream reads content of files with *xlsx* module `https://www.npmjs.com/package/xlsx` and meta data of the file using embedded node.js module *fs* and pushes it to the up stream.

The **function getFilesStream** creates readable stream for reading content of directory including nested directories and returns it. The **variable getDataStream** defined by stream in a object mode. For every file path it accepts from the down stream checks its extensions if its .xlsx then it reads it and obtains first sheet from the book, and reads meta data of the file and pushes it to the up stream. This two streams are piped in to combined stream using module *multipipe* `https://www.npmjs.com/package/multipipe` and returned by the function exported by this module. In other words, this module exports function which accepts path to the directory and returns array of deferred values each of which is a javascript object with following structure (Listing 7.2)

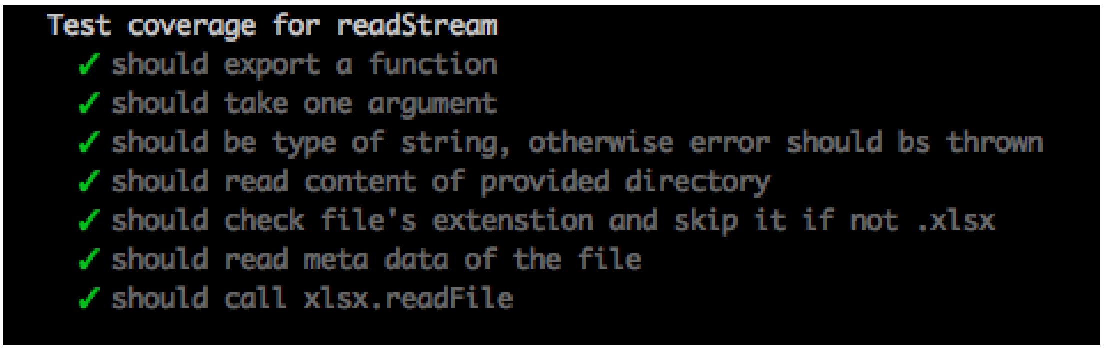
```
{
  fileName: <String>,
  meta: <Object>,
  sheet: <Object>
}
```

Listing 7.2: Read Stream Output / Transform Stream Input JSON

7.1.1 Test Coverage

Following test cases can show each step incrementally performed during the develop process. All calls to systems I/O (e. g. read file meta data, read file) are implemented using stubs. **Stub** is an implementation of *proxy pattern* which allows to redefine functions behavior. This gives an opportunity to improve speed of tests by replacing functions with call to I/O with empty functions.

The result of tests execution for this module is following (Fig.: 7.2):



```
Test coverage for readStream
  ✓ should export a function
  ✓ should take one argument
  ✓ should be type of string, otherwise error should bs thrown
  ✓ should read content of provided directory
  ✓ should check file's extenstion and skip it if not .xlsx
  ✓ should read meta data of the file
  ✓ should call xlsx.readFile
```

Figure 7.2: Test coverage for read_stream.js

7.1.2 Correspondence to design principles

- Closing - module exports single function which returns stream.
- Code to an Interface - calls are done to the public functions of required objects.
- Do not repeat yourself - no code duplication;
- The Single-Responsibility Principle - can be changed only due to the change of input type (e. g. another spreadsheet type)
- The Open/Closed Principle - new pipes can be added in a single place;
- The Liskov Substitution Principle - no inheritance;
- The Dependency-Inversion Principle -
  - Higher level module *index.js* does not depend on implementation current library
  - Current library depends on the input type provided by the *index.js*
- The Interface Segregation Principle - no calls of unnecessary methods;
- Principles of Least Knowledge - communication done only with xlsx files;
- Principle of Loose Coupling - communication is done via standard file system and standard stream interfaces.

7.2 Transform Stream

This part of the system performs translation of the Test Sheet in to executable javascript code. In this realization of the Test Sheet concept translation performed in a four stages.

```
var through = require('through2');
var template = require('../template');
var scheme = require('../scheme');

function transform(data, enc, callback) {
  var generalScheme = scheme.scheme.createScheme(data.sheet);
  var orderScheme = scheme.order.makeOrder(data.sheet, generalScheme
      );
  var executionScheme = scheme.executionScheme.create(generalScheme,
      orderScheme);
  var res = {
    fileName: data.fileName,
    meta: data.meta,
    content: template.applyTemplate(
        data.sheet,
        executionScheme,
        data.fileName),
  };

  this.push(res);
  callback();
};
```

Listing 7.3: Transform Stream

The first stage necessary for the translation is a creation of *schema* object of the Test Sheet. Next stage is detection of execution order. The third step is merging results from previous steps. The last stage is an application of a template to the content of Test Sheet together with the file name of the xlsx file and execution schema created on a previous stage. The result of this transformation is pulled by the downstream as a *content* property of an object together with *meta data* and *fileName* obtained from the upstream (Listing: 7.4).

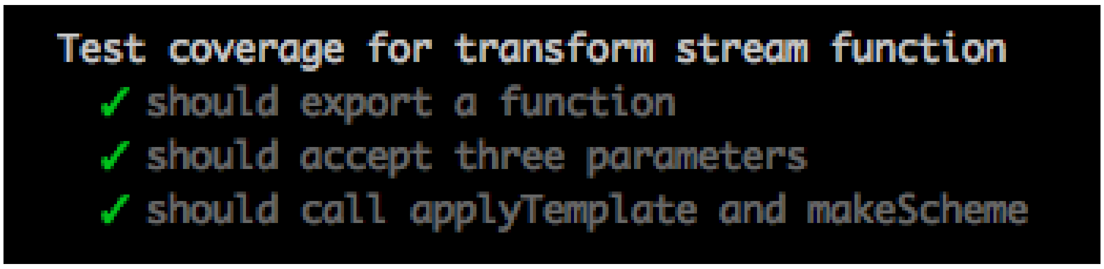
```
{
  fileName: <String>,
  meta: <Object>,
  contentent: <String>
}
```

Listing 7.4: Transform Stream Output / Write Stream Input JSON

7.2.1 Test Coverage

Following test cases can show each step incrementally performed during the develop process. All calls to systems I/O (e. g. read file meta data, read file) are implemented using stubs. **Stub** is an implementation of *proxy pattern* which allows to redefine functions behavior. This gives an opportunity to improve speed of tests by replacing functions with call to I/O with empty functions.

The result of test execution:



```
Test coverage for transform stream function
  ✓ should export a function
  ✓ should accept three parameters
  ✓ should call applyTemplate and makeScheme
```

Figure 7.3: Test coverage for transform_stream.js

7.2.2 Correspondence to design principles

- Closing - module exports stream object only;
- Code to an Interface - function invocation done via public functions of imported methods.
- Do not repeat yourself - no code duplication;
- The Single-Responsibility Principle - can be changed only due to the change of Test Sheet type;
- The Open/Closed Principle - new pipes can be added in a single place with adding new functions to be wrapped in them;
- The Liskov Substitution Principle - no inheritance;

- The Dependency-Inversion Principle -
  - Higher level module *index.js* does not depend on implementation current library;
  - Current library depends on the input type provided by the *index.js*;
- The Interface Segregation Principle - no calls of unnecessary methods;
- Principles of Least Knowledge - direct communication done only with helper functions for schema creation and content generation.
- Principle of Loose Coupling - communication is done via the standard stream interface

7.3 Schema

This part of the system creates a schema object from the Test Sheet object, the only parameter accepted by the main function of this module is a Test Sheet object (the content of xlsx file). Due to the structure of object was taken decision to perform creation of the schema in two stages.

The **first stage** is creation of javascript object with Test Sheet properties (e. g. *description, moduleUnderTest, objectsUnderTest, methodsUnderTest, inputs, outputs, invocations*) value of each property except *description* and *moduleUnderTest* are arrays of string with values which represent cell addresses for representative property of the Test Sheet. For the first two properties the values are strings with values (in a Test Sheet) from the cells *A1* and *A2* respectively(Listing: 7.5).

```
{
  description: 'Demonstaration',
  moduleUnderTest: 'stack',
  objectsUnderTest:[ 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'A9', 'A10'
      , 'A11'],
  methodsUnderTest: [ 'B3', 'B4', 'B5', 'B6', 'B7', 'B8', 'B9', 'B10
      ', 'B11'],
  inputs: ['C4', 'C6', 'C7', 'C9', 'C11'],
  outputs: [ 'F3', 'F4', 'F5', 'F6', 'F7', 'F8', 'F9', 'F10', 'F11'
      ],
  invocations: [ 'E3', 'E4', 'E5', 'E6', 'E7', 'E8', 'E9', 'E10', '
      E11'],
}
```

Listing 7.5: Result of the first stage of Test Sheet scheme creation

The **sectod stage** is a pivot transformation of the scheme provided by the result of previous stage. The output of this transformation is an object with properties as a numbers of rows in the Test Sheet object. The values for first and second properties are values of *description* and *module* under test from the respective Test Sheet. While all next are objects which consists of following Test Sheet properties: *objectUnderTest, methodUnderTest, inputs, outputs, invocations.* With values as single element arrays with coordinate of respective cell (Listing 7.6).

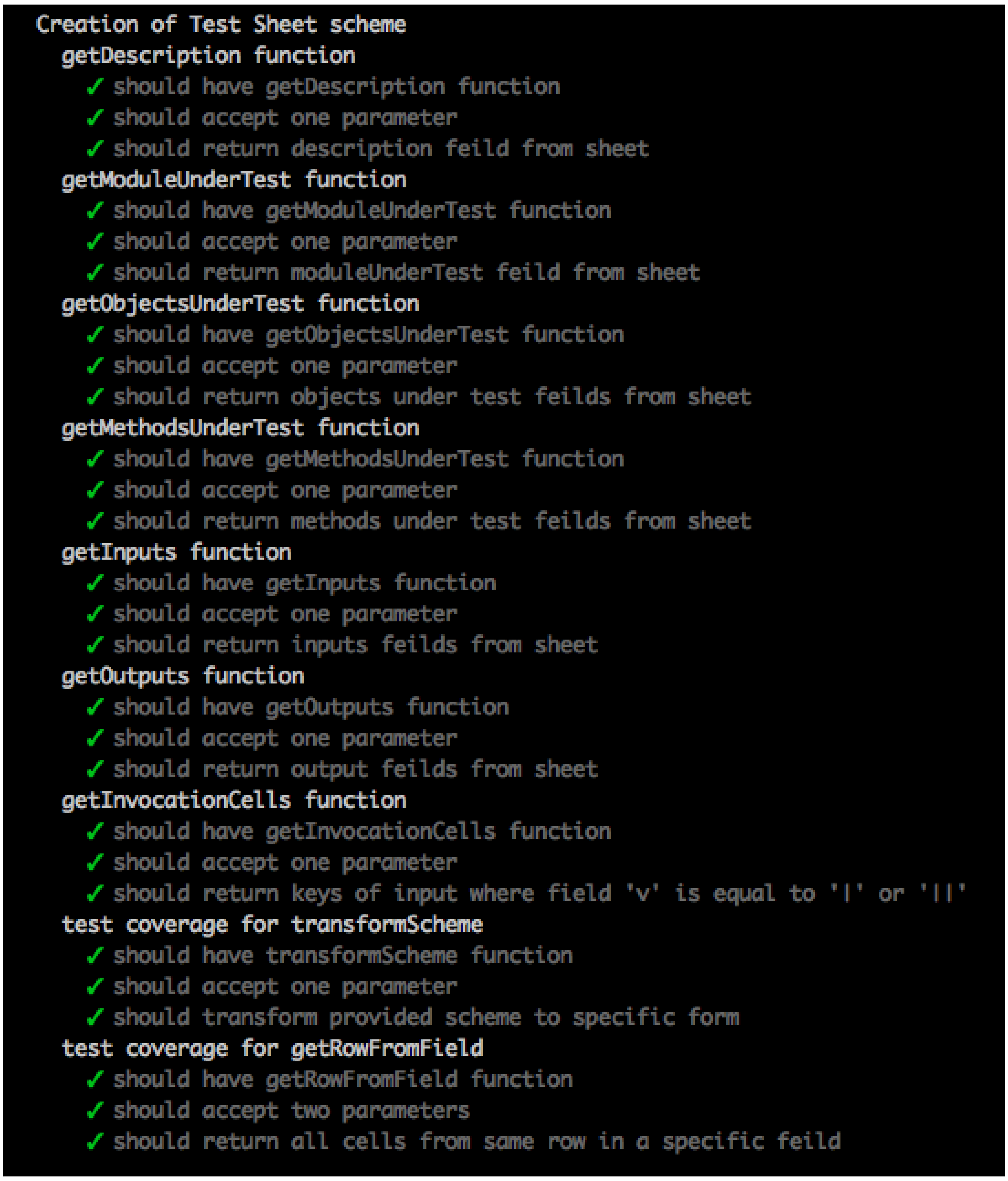
```
{
  "1":"Demonstaration",
  "2":"stack",
  "3":{"objectsUnderTest":["A3"],
        "methodsUnderTest":["B3"],
        "inputs":[],
        "outputs":["E3"],
        "invocations":["D3"]},
  "4":{"objectsUnderTest":["A4"],
       "methodsUnderTest":["B4"],
       "inputs":["C4"],
       "outputs":["E4"],
       "invocations":["D4"]},
  "5":{"objectsUnderTest":["A5"],
       "methodsUnderTest":["B5"],
       "inputs":[],
       "outputs":["E5"],
       "invocations":["D5"]},
  "6":{"objectsUnderTest":["A6"],
       "methodsUnderTest":["B6"],
       "inputs":["C6"],
       "outputs":["E6"],
       "invocations":["D6"]},
  "7":{"objectsUnderTest":["A7"],
       "methodsUnderTest":["B7"],
       "inputs":["C7"],
       "outputs":["E7"],
       "invocations":["D7"]},
  "8":{"objectsUnderTest":["A8"],
       "methodsUnderTest":["B8"],
       "inputs":[],
       "outputs":["E8"],
       "invocations":["D8"]},
  "9":{"objectsUnderTest":["A9"],
       "methodsUnderTest":["B9"],
       "inputs":["C9"],
       "outputs":["E9"],
       "invocations":["D9"]},
  "10":{"objectsUnderTest":["A10"],
        "methodsUnderTest":["B10"],
        "inputs":[],
        "outputs":["E10"],
        "invocations":["D10"]},
  "11":{"objectsUnderTest":["A11"],
        "methodsUnderTest":["B11"],
        "inputs":["C11"],
        "outputs":["E11"],
        "invocations":["D11"]}
}
```

Listing 7.6: Result of the scheme creation for Test Sheet

7.3.1 Test Coverage

Following test cases can show each step incrementally performed during the develop process. All calls to systems I/O (e. g. read file meta data, read file) are implemented using stubs. **Stub** is an implementation of *proxy pattern* which allows to redefine functions behavior. This gives an opportunity to improve speed of tests by replacing functions with call to I/O with empty functions.

The result of test execution:



```
 Creation of Test Sheet scheme
   getDescription function
     ✓ should have getDescription function
     ✓ should accept one parameter
     ✓ should return description feild from sheet
   getModuleUnderTest function
     ✓ should have getModuleUnderTest function
     ✓ should accept one parameter
     ✓ should return moduleUnderTest feild from sheet
   getObjectsUnderTest function
     ✓ should have getObjectsUnderTest function
     ✓ should accept one parameter
     ✓ should return objects under test feilds from sheet
   getMethodsUnderTest function
     ✓ should have getMethodsUnderTest function
     ✓ should accept one parameter
     ✓ should return methods under test feilds from sheet
   getInputs function
     ✓ should have getInputs function
     ✓ should accept one parameter
     ✓ should return inputs feilds from sheet
   getOutputs function
     ✓ should have getOutputs function
     ✓ should accept one parameter
     ✓ should return output feilds from sheet
   getInvocationCells function
     ✓ should have getInvocationCells function
     ✓ should accept one parameter
     ✓ should return keys of input where field 'v' is equal to '|' or '||'
   test coverage for transformScheme
     ✓ should have transformScheme function
     ✓ should accept one parameter
     ✓ should transform provided scheme to specific form
   test coverage for getRowFromField
     ✓ should have getRowFromField function
     ✓ should accept two parameters
     ✓ should return all cells from same row in a specific feild
```

Figure 7.4: Test coverage for scheme.js

7.3.2 Correspondence to design principles

- Closing - comparison type is closed within the *getInvocationCells* function.
- Code to an Interface - exports single function.
- Do not repeat yourself - no code repetition .

- The Single-Responsibility Principle - no reason to be changed
- The Open/Closed Principle - violated due to the two types comparison;
- The Liskov Substitution Principle - no inheritance.
- The Dependency-Inversion Principle - does not have any lower level modules;
- The Interface Segregation Principle - does not have any lower level modules;
- Principles of Least Knowledge - can be invoked only via direct import;
- Principle of Loose Coupling - exports single function;

7.4 Execution Order

This part of the system is responsible for definition of test steps execution. In general case the execution order of tests within a Basic Test Sheet is defined by the order of test steps. Further all Test Sheets support references which provides an ability to define input for one test step as an output of some of the previous steps.

However, in *asynchronous* software the moment when execution is completed comes after the period of time when next function is invoked. Moreover, *real-time* systems have time constraints on their execution together with non-deterministic behavior. This two facts forces to define moment of each test step execution depending on the source of their input parameters.

For definition of test steps execution order was taken decision to separate test steps in two types. *The first* type covers test steps whose behavior does not make influence to next steps (do not define input parameters for further test steps) and is not determined by any of previous test steps. *The second* type covers test steps which execution result determines behavior of next test steps or defined by previous steps.

The process of order definition is following. Main function accepts two input parameters: *sheet* - the content of xlsx file which defines test sheet and *scheme* - the object generated by the main function from *scheme.js* file (Listing: 7.7). Walking through the scheme elements stating with the third (skipping service rows) performs check for each cell in a sheet object for determining is behavior of test step defined by result of any previous test step. In this case such test step will be pushed to the *chain* array as the first element and all dependent test steps will be pushed to the nested array as a list of dependent test steps. If *chain* is not empty it will be pushed to global (in a function scope) *chains* array. All other rows will be pushed to the *linear array*. Such approach guarantees saving order of rows and creates a record of tree as a list of direct child nodes.

```
function makeOrder(sheet, scheme) {
  var chains = [];
  var linear = [];

  for(var i = 3; i < Object.keys(scheme).length; i++){
    var chain = [];

    for (var cell in sheet) {
      if (sheet[cell].f == scheme[i].outputs[0]) {
        if(chain[0]){
          chain[1].push(parseInt(cell.slice(1)));
        } else {
          chain[0] = parseInt(sheet[cell].f.slice(1));
          chain[1] = [parseInt(cell.slice(1))];
        };
      }
    };
    if(chain.length > 0)
      chains.push(chain);
  };

  for(var i = 1; i < Object.keys(scheme).length; i++){
    if (!isIn(i, chains)) linear.push(i);
  };

  return(linear.concat(transform(chains)));
};
```

Listing 7.7: Generation of tree structure as a list of childnodes

For the further creation of test steps execution order the list of direct child nodes should be merged in to nested list for child nodes which has other child nodes. In terms of test steps this means that result one test step can define behavior of few test steps either directly or via some intermediate test steps. The merging process is performed by the *transformation* function (Listing 7.8). It accepts an array object (*chains* array described previously) and performs iterative walk through with recursive call for each nested array. During each iteration this function iterates through values of childes (nested arrays) and checks is any of them is a parent for other childes (is it present as a first element in next arrays). In such case it takes list of its childes and creates a two dimensional array which replaces the the parent element in a list of childes with removing parent entry together with its childes from the current position. This operation of replacement performed recursively through all nested arrays using *isIn* function which determines is given element a belongs to the provided array including nested arrays.

```
function transform(array) {
  for (var i = 0; i < array.length; i++) {
    if (array[i][0] && array[i][1]) {
      for (var j = 0; j < array[i][1].length; j++) {
        for (var k = i + 1; k < array.length; k++) {
          if (array[i][1][j] == array[k][0]) {
            array[i][1][j] = [array[i][1][j], array[k][1]];
            array.splice(k, 1);
          };
        };
      };
    } else if(Array.isArray(array[0])) {
      if (isIn(array[i], array[0])) {
        array.splice(i, 1);
        i--;
      };
    };
    if(Array.isArray(array[i])){
      for (var j = 0; j < array[i].length; j++) {
        if(Array.isArray(array[i][j]))
          transform(array[i][j]);
      };
    };
  };
  return array;
};
```

Listing 7.8: Generation of nested data structure

The final step is concatenation of arrays from *first* and *second* test steps types.

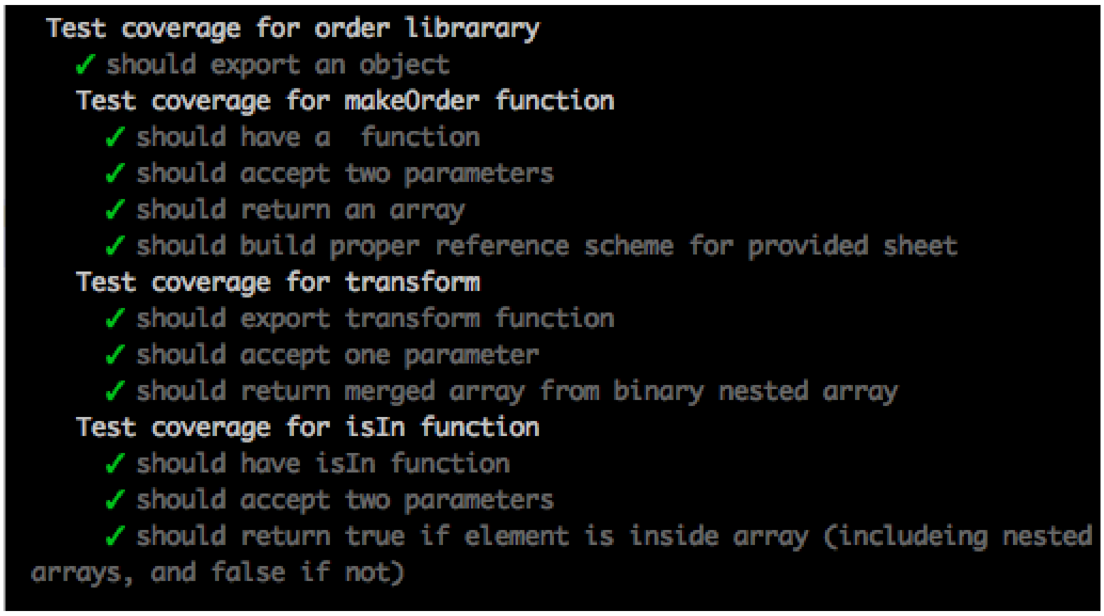
For our example, rows *one* and *two* belong to the first type as a service rows which does not define any test steps together with rows *three* and *four*. Note that input parameter for the test step defined it a row four is referenced by the expected output of the test step defined in a row five. However, the input parameter can not be changed by the execution process of test step and can be passed to the next step before starting execution without any execution logic violation. In the same time result of test step *five* is an input parameter for the steps *six*, *seven* and *ten* which in turn defines an input for the test step *eleven* (Listing: 7.9).

```
[
  1,
  2,
  3,
  4,
  [5,[6,7,[10,[11]]]],
  [8,[9]]
]
```

Listing 7.9: Execution order

7.4.1 Test coverage

Following test cases can show each step incrementally performed during the develop process.



```
  Test coverage for order librarary
    ✓ should export an object
    Test coverage for makeOrder function
      ✓ should have a  function
      ✓ should accept two parameters
      ✓ should return an array
      ✓ should build proper reference scheme for provided sheet
    Test coverage for transform
      ✓ should export transform function
      ✓ should accept one parameter
      ✓ should return merged array from binary nested array
    Test coverage for isIn function
      ✓ should have isIn function
      ✓ should accept two parameters
      ✓ should return true if element is inside array (includeing nested
 arrays, and false if not)
```

Figure 7.5: Test coverage for order.js

7.4.2 Correspondence to design principles

- Closing - nothing is likely to change;
- Code to an Interface - exports a single function;
- Do not repeat yourself - no code duplication;
- The Single-Responsibility Principle - nothing is likely to change;
- The Open/Closed Principle - no reason to change, no need for a new functionality;
- The Liskov Substitution Principle - no inheritance;
- The Dependency-Inversion Principle - does not have any lower level modules;
- The Interface Segregation Principle - does not have any lower level modules;
- Principles of Least Knowledge - can be invoked only via direct import;
- Principle of Loose Coupling - exports a single function.

7.5 Execution Schema

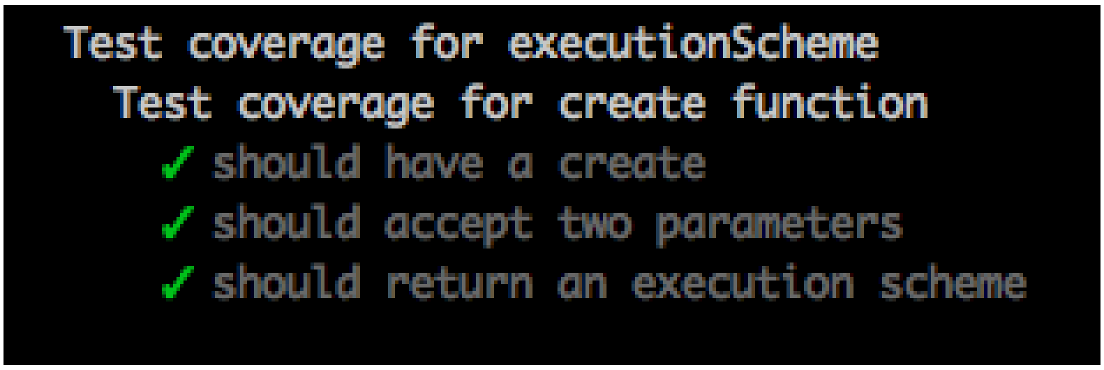
In this step performed recursive replacement of respective elements in an *execution order* array with elements from the *scheme* object. The function (Listing: 7.10) accepts two arguments: *scheme* object and *order* array. If an element is of an input array is an array it performs recursive call.

```
function create(scheme, order) {
  var result = order.slice();
  for (line in order) {
    if (Array.isArray(order[line])) {
      result[line] = create(scheme, order[line]);
    } else {
      result[line] = scheme[order[line]];
    }
  };
  return result;
};
```

Listing 7.10: Result of the scheme creation for Test Sheet

7.5.1 Test Coverage

Following test cases can show each step incrementally performed during the develop process.



```
Test coverage for executionScheme
  Test coverage for create function
    ✓ should have a create
    ✓ should accept two parameters
    ✓ should return an execution scheme
```

Figure 7.6: Test coverage for creation of execution scheme

7.5.2 Correspondence to design principles

- Closing - nothing is likely to change;

- Code to an Interface - exports a single function;
- Do not repeat yourself - no code duplication;
- The Single-Responsibility Principle - nothing is likely to change;
- The Open/Closed Principle - no reason to change, no need for a new functionality;
- The Liskov Substitution Principle - no inheritance;
- The Dependency-Inversion Principle - does not have any lower level modules;
- The Interface Segregation Principle - does not have any lower level modules;
- Principles of Least Knowledge - can be invoked only via direct import;
- Principle of Loose Coupling - exports a single function.

7.6 Template

The translation to executable javascript file is made by apply the *template* function. This function accepts three parameters *sheet ¡Object¿, scheme ¡Object¿, fileName¡String¿*. *Sheet* is a javascript object returned by reading the content of Test Sheet file. The *scheme* is an execution order scheme. The *fileName* is a path to the TestSheet file. This function creates a string content for javascript file based on provided parameters. The creation takes part in a four stages.

The **first stage** is adding a description from the first element of the scheme object. It enclosed within the multiline comment symbols.

The **second stage** is adding require of reporting module for representation of execution of the tests result.

The **third stage** is adding declarations of the module scope variables. The variable names are taken from the coordinates of cells which store input and output parameters in a Test Sheet. The values of this variables are taken as from the cells with respective addresses.

The **fourth stage** is adding calls of the methods (Listing: **??**). This stage is the most important part of this module. Since there are two types of execution orders adding function calls performed in a two steps.

The ***first step*** is adding calls for independent test steps. This is made be iterative call of *makeCall* function with appending closing brackets to the end of the result string during each iteration. This function simply performs mapping between *sheet* object (content of xlsx file), *scheme* entry (test step) and string with content for executable javascript file (Listing: 7.11).

```
function makeCall(sheet, scheme, indent, res, fileName) {
  return res.concat('  '.repeat(indent) + scheme.objectsUnderTest
      [0],
    '.' + getValue(sheet, scheme.methodsUnderTest),
    '.call(' + scheme.objectsUnderTest[0] + getCallInputs(sheet,
      scheme)
      + ', function(err, data){\n'
      + '  '.repeat(indent + 1)
      + 'if(err)'
      + '  ' + 'return(err, null);\n'
      + '  '.repeat(indent + 1)
      + 'makeComparisonAndWriteResult('
      + scheme.outputs
      + ', data, '
      + '\'' + getValue(sheet, scheme.invocations, true) + '\'' + ',
        '
      + '\'' + getValue(sheet, scheme.objectsUnderTest) + ' ' +
        getValue(sheet, scheme.methodsUnderTest)+ '\', ' + '\'' +
        fileName +'\', '+ '\'' + scheme.outputs + '\'' + ');\n'
      + '  '.repeat(indent + 1)
      + scheme.outputs
      + ' = data;\n');
};
```

Listing 7.11: Function makeCall from template module

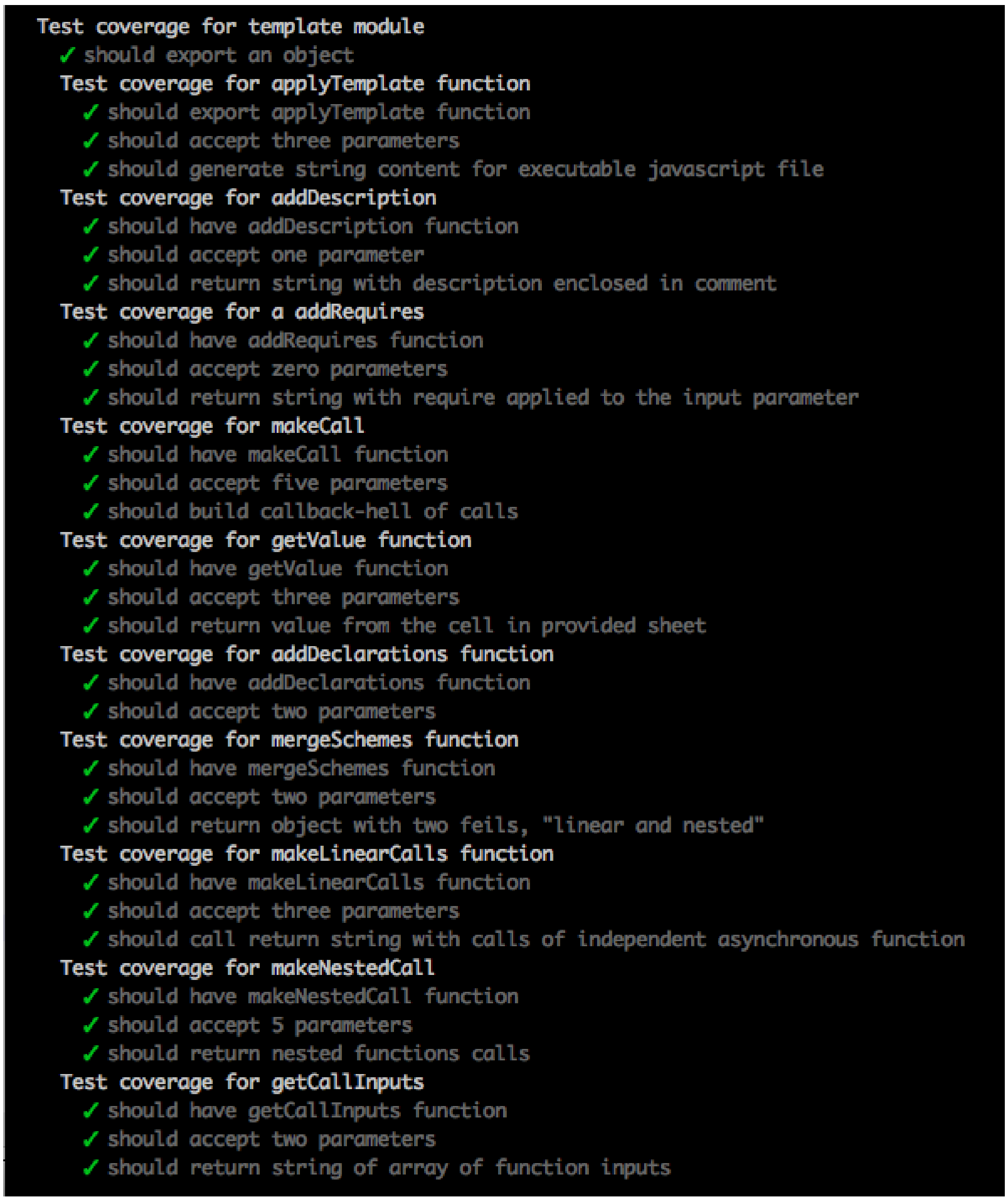
Note that functions in the generated code are not called directly but via *call* method of the function's prototype. This method provides an opportunity to call the function from different context (environment). In current example the implementation of *stack* module does not depend on the environment in which it was called. But in some cases the result of function's invocation can depend on the environment in which it was invoked. One of such cases can be creation of child process. The environment for such calls should be passed as a *obcet under test*.

The ***second step*** is adding calls for interdependent test steps. For every element of *nestedCalls* array it calls *makeCall* function and if next element is an array this functions performs recursive call with appending of the closing brackets to the result of this call. In case if next element is not an array it appends closing brackets and performs next iteration.

7.6.1 Test Coverage

Following test cases can show each step incrementally performed during the develop process.

The result of test execution:



```
 Test coverage for template module
   ✓ should export an object
   Test coverage for applyTemplate function
     ✓ should export applyTemplate function
     ✓ should accept three parameters
     ✓ should generate string content for executable javascript file
   Test coverage for addDescription
     ✓ should have addDescription function
     ✓ should accept one parameter
     ✓ should return string with description enclosed in comment
   Test coverage for a addRequires
     ✓ should have addRequires function
     ✓ should accept zero parameters
     ✓ should return string with require applied to the input parameter
   Test coverage for makeCall
     ✓ should have makeCall function
     ✓ should accept five parameters
     ✓ should build callback-hell of calls
   Test coverage for getValue function
     ✓ should have getValue function
     ✓ should accept three parameters
     ✓ should return value from the cell in provided sheet
   Test coverage for addDeclarations function
     ✓ should have addDeclarations function
     ✓ should accept two parameters
   Test coverage for mergeSchemes function
     ✓ should have mergeSchemes function
     ✓ should accept two parameters
     ✓ should return object with two feils, "linear and nested"
   Test coverage for makeLinearCalls function
     ✓ should have makeLinearCalls function
     ✓ should accept three parameters
     ✓ should call return string with calls of independent asynchronous function
   Test coverage for makeNestedCall
     ✓ should have makeNestedCall function
     ✓ should accept 5 parameters
     ✓ should return nested functions calls
   Test coverage for getCallInputs
     ✓ should have getCallInputs function
     ✓ should accept two parameters
     ✓ should return string of array of function inputs
```

Figure 7.7: Test coverage for template.js

7.6.2 Correspondence to design principles

- Closing - content generation is enclosed within three independent functions;
- Code to an Interface - exports a single function;

- Do not repeat yourself - no code duplication;
- The Single-Responsibility Principle - can be changed only due to the requirements for generated code;
- The Open/Closed Principle - no need to be changed;
- The Liskov Substitution Principle - no inheritance;
- The Dependency-Inversion Principle - does not have any lower level dependencies;
- The Interface Segregation Principle - does not have any lower level dependencies;
- Principles of Least Knowledge - can be invoked only via direct import;
- Principle of Loose Coupling - exports a single function.

7.7 Write Stream

This part of the system implements on function and exports it wrapped in to object stream. It receives an object from the up stream with following structure (Lisitng: 7.4). It tries to read meta data of the file with same name as Test Sheet file but with the .js extension, it possible only if the file exists. In other case, the exception will be thrown and caught by catch block which implements creation of the file with content received from the up stream, and notification of user about successfully created file via standard system output or an error if the file failed to create. If file already exists the function will be able to read its meta data. After it, this meta data will be compared with meta data of the Test Sheet file received from the upstream. In case if modification date of the Test Sheet file is bigger then modification data of javascript file system will overwrite the javascript file with content received from the upstream. In other case case (Tests sheet file was not updated after creation of representative javascript file) the system will switch to the next object received from the up stream. The result javascript file generated from the provided Test Sheet looks as following (Listing: 7.12)

```
/*
 Demonstaration
 */
var makeComparisonAndWriteResult = require('../compare_and_write');

var A3 = require('./stack');
var E3 = {"size": 0};
var A4 = require('./stack');
var C4 = {"el": 1};
var E4 = {};
var A5 = require('./stack');
var E5 = C4;
var A6 = require('./stack');
var C6 = E5;
var E6 = {};
var A7 = require('./stack');
var C7 = E5;
var E7 = {};
var A10 = require('./stack');
var E10 = E5;
var A11 = require('./stack');
var C11 = E10;
var E11 = {};
var A8 = require('./stack');
var E8 = {"el": 5};
var A9 = require('./stack');
var C9 = E8;
var E9 = {};


A3.size.call(this, function(err, data) {
  if (err) return (err, null);
  makeComparisonAndWriteResult(E3, data, '||', 'stack size', 'demo/
      demo.xlsx', 'E3');
  E3 = data;
});
A4.push.call(this, {
  "el": 1
}, function(err, data) {
  if (err) return (err, null);
  makeComparisonAndWriteResult(E4, data, '||', 'stack push', 'demo/
      demo.xlsx', 'E4');
  E4 = data;
});
```

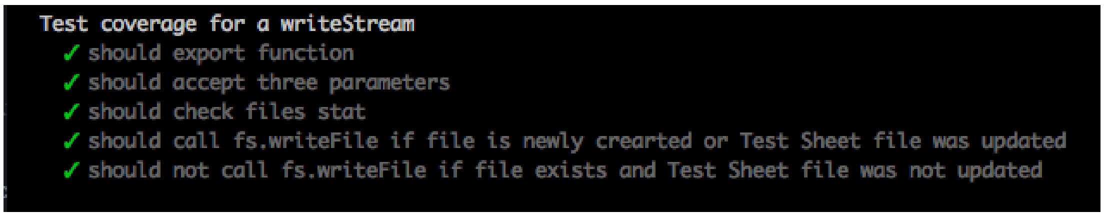
```
A5.top.call(this, function(err, data) {
  if (err) return (err, null);
  makeComparisonAndWriteResult(E5, data, '||', 'stack top', 'demo/
     demo.xlsx', 'E5');
  E5 = data;
  A6.push.call(this, E5, function(err, data) {
    if (err) return (err, null);
    makeComparisonAndWriteResult(E6, data, '|', 'stack push', 'demo/
       demo.xlsx', 'E6');
    E6 = data;
  });
  A7.push.call(this, E5, function(err, data) {
    if (err) return (err, null);
    makeComparisonAndWriteResult(E7, data, '|', 'stack push', 'demo/
       demo.xlsx', 'E7');
    E7 = data;
    A10.pop.call(this, function(err, data) {
      if (err) return (err, null);
      makeComparisonAndWriteResult(E10, data, '||', 'stack pop', '
         demo/demo.xlsx', 'E10');
      E10 = data;
      A11.push.call(this, E10, function(err, data) {
        if (err) return (err, null);
        makeComparisonAndWriteResult(E11, data, '||', 'stack push',
           'demo/demo.xlsx', 'E11');
        E11 = data;
      });
    });
  });
});
A8.pop.call(this, function(err, data) {
  if (err) return (err, null);
  makeComparisonAndWriteResult(E8, data, '||', 'stack pop', 'demo/
     demo.xlsx', 'E8');
  E8 = data;
  A9.push.call(this, E8, function(err, data) {
    if (err) return (err, null);
    makeComparisonAndWriteResult(E9, data, '||', 'stack push', 'demo
       /demo.xlsx', 'E9');
    E9 = data;
  });
});
```

Listing 7.12: Generated javascript file

7.7.1 Test Coverage

Following test cases can show each step incrementally performed during the develop process. All calls to systems I/O (e. g. read file meta data, read file) are implemented using stubs.

The result of tests execution for this module is following (Fig.: 7.8):



```
Test coverage for a writeStream
  ✓ should export function
  ✓ should accept three parameters
  ✓ should check files stat
  ✓ should call fs.writeFile if file is newly crearted or Test Sheet file was updated
  ✓ should not call fs.writeFile if file exists and Test Sheet file was not updated
```

Figure 7.8: Test coverage for write_stream.js

7.7.2 Correspondence to design principles

- Closing - module exports single stream object with wrapped function.
- Code to an Interface - calls are done to the public functions of required objects.
- Do not repeat yourself - small code duplication in callbacks for file writing, avoiding which will add more complexity;
- The Single-Responsibility Principle - can be changed only due to the change of output type (e. g. another output file extension)
- The Open/Closed Principle - performs writing to the file only;
- The Liskov Substitution Principle - no inheritance;
- The Dependency-Inversion Principle -
  - Higher level module *index.js* does not depend on implementation current library
  - Current library depends on the input type provided by the *index.js*

- The Interface Segregation Principle - no calls of unnecessary methods;
- Principles of Least Knowledge - communication done only with js files;
- Principle of Loose Coupling - communication is done via standard file system and standard stream interfaces.

7.8 Reporting mechanism

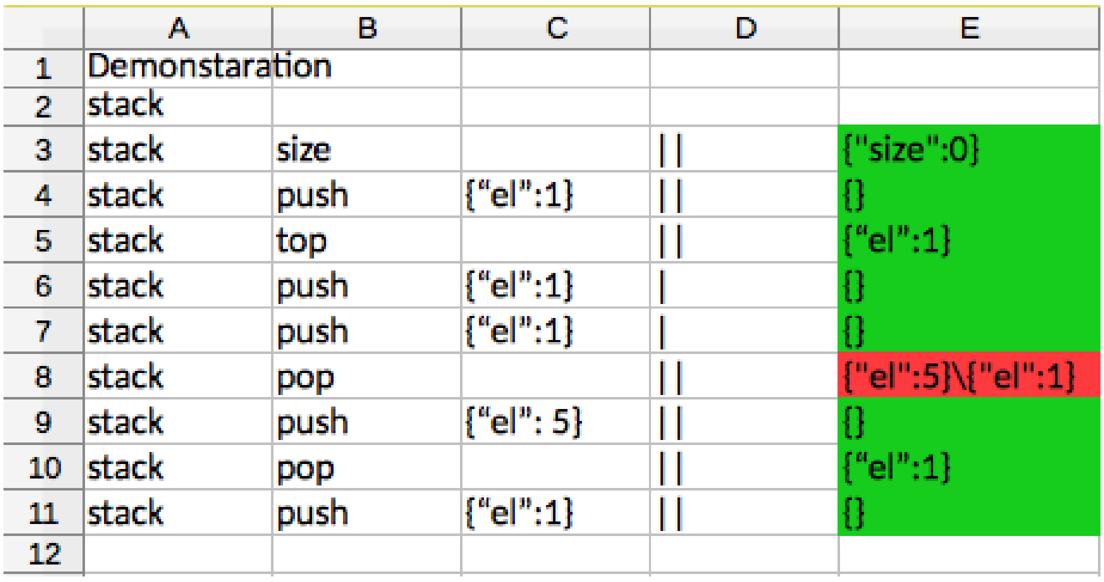
The reporting mechanism made as a standalone npm package which must me installed globally for being available at any place of the system where it is called by generated javascript files described in a previous section. The folder/file structure of the application looks as following:

- index.js
- package.json
- ReadMe.md
- lib/
  - compare_and_write.js
- test/
  - compare_and_write.js
- node_modules/

7.8.1 Implementation

```
function makeComparisonAndWriteResult(expected, returned, deepness,
    scriptName, file, variable){
  var result = compare(expected, returned, deepness);
  var res = "";
  result ? res = "#00FF00" : res = "#FFFF00";
  var testSheet = xlsx.readFile(file, {'cellStyles': true});
  if(!result){
    testSheet.Sheets.Sheet1[variable].v = JSON.stringify(expected) +
        '\\' + JSON.stringify(returned);
    xlsx.writeFile(testSheet, file, {'cellStyles': true});
  };
  console.log(expected, returned, deepness, result);

  return(result);
};
```

Listing 7.13: Report mechanism's main function

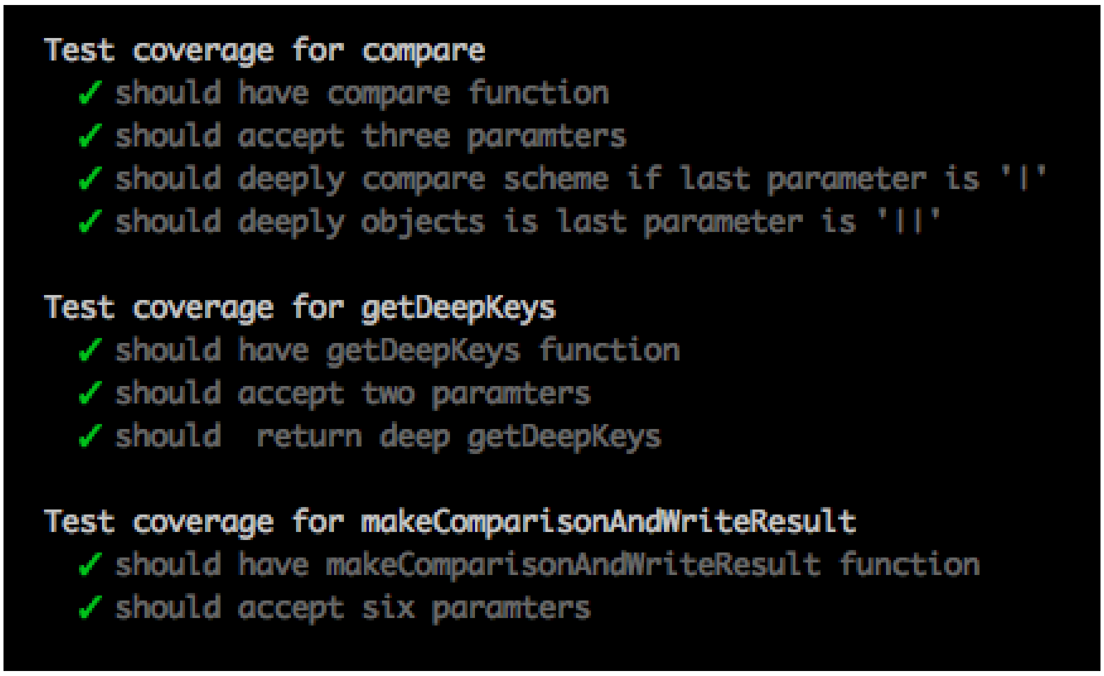


	A	B	C	D	E
1	Demonstaration				
2	stack				
3	stack	size		| |	{"size":0}
4	stack	push	{“el”:1}	| |	{}
5	stack	top		| |	{“el”:1}
6	stack	push	{“el”:1}	|	{}
7	stack	push	{“el”:1}	|	{}
8	stack	pop		| |	{"el":5}\{"el":1}
9	stack	push	{“el”: 5}	| |	{}
10	stack	pop		| |	{“el”:1}
11	stack	push	{“el”:1}	| |	{}
12					

Figure 7.9: Result Test Sheet for stack.js

7.8.2 Test coverage

Following test cases can show each step incrementally performed during the develop process. All calls to systems I/O (e. g. read file meta data, read file) are implemented using stubs. The result of tests execution for this module is following (Fig.: 7.10):



```
Test coverage for compare
  ✓ should have compare function
  ✓ should accept three paramters
  ✓ should deeply compare scheme if last parameter is '|'
  ✓ should deeply objects is last parameter is '||'

Test coverage for getDeepKeys
  ✓ should have getDeepKeys function
  ✓ should accept two paramters
  ✓ should  return deep getDeepKeys

Test coverage for makeComparisonAndWriteResult
  ✓ should have makeComparisonAndWriteResult function
  ✓ should accept six paramters
```

Figure 7.10: Test coverage for report mechanism

7.8.3 Correspondence to design principles

- Closing - comparison and reporting are done in a different independent functions
- Code to an Interface - exports a single function;
- Do not repeat yourself - no code duplication;
- The Single-Responsibility Principle - can be changed only due to the reporting media;
- The Open/Closed Principle - new functionality can be added as another exported function;
- The Liskov Substitution Principle - no inheritance (required as an npm package);
- The Dependency-Inversion Principle - requires standard *assert*, *files system* modules and *xlsx* module;
- The Interface Segregation Principle - does not have any lower level modules;
- Principles of Least Knowledge - can be invoked only via direct import;

- Principle of Loose Coupling - exports a single function.

Bibliography

[1] About node.js®. https://nodejs.org/en/about/.

[2] Analysis of generators and other async patterns in node. https://spion.github.io/posts/analysis-generators-and-other-async-patterns-node.html.

[3] Are you screen scraping or data mining? http://www.connotate.com/are-you-screen-scraping-or-data-mining/.

[4] The bank business model for apis: Identity. http://tomorrowstransactions.com/2015/03/the-bank-business-model-for-apis-identity/.

[5] Banks in germany. http://banksgermany.com/.

[6] Casperjs. http://casperjs.org/.

[7] Cucumber: Reference. https://cucumber.io/docs/reference.

[8] Eur-lex - 32015l2366 - en - eur-lex. http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32015L2366s.

[9] Events node.js. https://nodejs.org/api/events.html.

[10] figo — angel list. https://angel.co/figo-1.

[11] figo — crunchbase. https://www.crunchbase.com/organization/figo#/entity.

[12] Fints - wikipedia, free encyclopedia. https://en.wikipedia.org/wiki/FinTS.

[13] Fit. http://fit.c2.com/.

[14] Fit. http://fit.c2.com/wiki.cgi?IntroductionToFit.

[15] glog.isz.me - designing apis for asynchrony. http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony.

[16] Ich möchte mehr zu eurer vision erfahren! https://figo.zendesk.com/hc/de/articles/200974801-Ich-m%C3%B6chte-mehr-zu-eurer-Vision-erfahren-.

[17] Inheritance and the prototype chain. https://developer.mozilla.org/en/docs/Web/JavaScript/Inheritance_and_the_prototype_chain.

[18] Introduction to asynchronous programming. http://cs.brown.edu/courses/cs168/s12/handouts/async.pdf.

[19] An introduction to libuv: Basics of libuv. http://nikhilm.github.io/uvbook/basics.html.

[20] Introduction to object-oriented javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript.

[21] kriskowal/gtor: A general theory of reactivity. https://github.com/kriskowal/gtor/.

[22] Lars markull's answer to what are the benefits of an api for a consumer bank? - quora. https://www.quora.com/What-are-the-benefits-of-an-API-for-a-consumer-bank/answer/Lars-Markull?srid=hmj2&share=267222bd.

[23] Module counts. http://www.modulecounts.com/.

[24] Open banking apis: Threat and opportunity — consult hyperion. http://www.chyp.com/open-banking-apis-threat-and-opportunity/.

[25] The real cost of software errors. http://hdl.handle.net/1721.1/74607.

[26] Test sheets. http://swt.informatik.uni-mannheim.de/de/research/research-topics/test-sheets/.

[27] Test sheets. http://swt.informatik.uni-mannheim.de/de/research/research-topics/test-sheets/basic-test-sheets/.

[28] W3c document object notation. https://www.w3.org/DOM/#what.

[29] Was ist figo und was macht ihr? https://figo.zendesk.com/hc/de/articles/200862101-Was-ist-figo-und-was-macht-ihr-.

[30] Web scraping - wikipedia free encyclopedia. https://en.wikipedia.org/wiki/Web_scraping.

[31] Welcome to the libuv api documentation. `http://docs.libuv.org/en/v1.x/#features`.

[32] Wer sind eure partner? wie kann ich figo dort nutzen? `https://figo.zendesk.com/hc/de/articles/200907292-Wer-sind-eure-Partner-Wie-kann-ich-figo-dort-nutzen-`.

[33] Why i am switching to promises. `https://spion.github.io/posts/why-i-am-switching-to-promises.html`.

[34] Xunit. `http://www.martinfowler.com/bliki/Xunit.html`.

[35] xunit - wikipedia, free encyclopedia. `https://en.wikipedia.org/wiki/XUnit`.

[36] Colin Atkinson. L1-introduction. 2015.

[37] Colin Atkinson. L2-testingintroduction. 2015.

[38] Mario Casciaro. *NodeJS Design Patterns*. Packt Publishing, 2014.

[39] Catalin Cimpanu. Annoyed developer brings down thousands of javascript & node.js projects.

[40] Catalin Cimpanu. Node.js package manager vulnerable to malicious worm packages.

[41] John Dooley. *Software Development and Professional Practice*. Apress, Berkely, CA, USA, 1st edition, 2011.

[42] Robert L. Glass. Real-time: The &ldquo;lost world&rdquo; of software debugging and testing. *Commun. ACM*, 23(5):264–271, May 1980.

[43] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[44] Gerard Meszaros. *xUnit test patterns refactoring test code*. Safari Books Online. Addison-Wesley, Upper Saddle River, N.J., 2007.

[45] Sam Saccone. npm hydra worm disclosure. 2016.

[46] J. J. P. Tsai, K. Y. Fang, and Y. D. Bi. On real-time software testing and debugging. In *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International*, pages 512–518, Oct 1990.

Appendix

Eidesstattliche Erklärung

Hiermit versichere ich, dass diese Abschlussarbeit von mir persönlich verfasst ist und dass ich keinerlei fremde Hilfe in Anspruch genommen habe. Ebenso versichere ich, dass diese Arbeit oder Teile daraus weder von mir selbst noch von anderen als Leistungsnachweise andernorts eingereicht wurden. Wörtliche oder sinngemäße Übernahmen aus anderen Schriften und Veröffentlichungen in gedruckter oder elektronischer Form sind gekennzeichnet. Sämtliche Sekundärliteratur und sonstige Quellen sind nachgewiesen und in der Bibliographie aufgeführt. Das Gleiche gilt für graphische Darstellungen und Bilder sowie für alle Internet-Quellen.

Ich bin ferner damit einverstanden, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs in elektronischer Form anonymisiert versendet und gespeichert werden kann. Mir ist bekannt, dass von der Korrektur der Arbeit abgesehen werden kann, wenn die Erklärung nicht erteilt wird.

Mannheim, April 2, 2016 Unterschrift

Abtretungserklärung

Hinsichtlich meiner Studienarbeit/Bachelor-Abschlussarbeit/Diplomarbeit räume ich der Universität Mannheim/Lehrstuhl für Softwaretechnik, Prof. Dr. Colin Atkinson, umfassende, ausschließliche unbefristete und unbeschränkte Nutzungsrechte an den entstandenen Arbeitsergebnissen ein.

Die Abtretung umfasst das Recht auf Nutzung der Arbeitsergebnisse in Forschung und Lehre, das Recht der Vervielfältigung, Verbreitung und Übersetzung sowie das Recht zur Bearbeitung und Änderung inklusive Nutzung der dabei entstehenden Ergebnisse, sowie das Recht zur Weiterübertragung auf Dritte.

Solange von mir erstellte Ergebnisse in der ursprünglichen oder in überarbeiteter Form verwendet werden, werde ich nach Maßgabe des Urheberrechts als Co-Autor namentlich genannt. Eine gewerbliche Nutzung ist von dieser Abtretung nicht mit umfasst.

Mannheim, April 2, 2016 Unterschrift