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

Acknowledgments

Abstract

Despite the big amount of domain specific languages and automated code generation tools modern approaches to the test definition and execution still have a high entrance level for people without experience in software development. Further, some of the existing approaches provide an opportunity to define tests easily, however they still require developers to write code which will transform defined tests into executable code.

Test Sheets is an approach for test definitions created by Software Engineering group of the University of Mannheim. It provides tool-free user experience of test definitions via the use of regular spreadsheets editors. As a result, it reduces involvement of technical staff into this process to minimum.

This paper describes the process of development and use of system which implements Basic Test Sheets in case of figo GmbH. The system design was made with taking in to account next requirements. First, it must be used for asynchronous testing. Second, the system under test represents a real-time software. This two factors leads to the case when one test step can be started before the completion of previous steps while exact time required for test step completion is critical but not known.

The software system implemented within the scope of this research shows the possibility of usage of Test Sheets for asynchronous testing of real-time software.

The research results stated in this paper cover following aspects. System's performance measurements during the code generation and execution. The fit of the system to the indicated problem. User experience analysis after pilot stage of the project. The misfits inherent to the implemented software. The benefits of the system use from different perspectives.

Contents

Contents   ix

List of Figures

List of Tables

1 Introduction

This paper shows the case of the Test Sheets' use for asynchronous testing of a real-time software. Tests definition using Test Sheets is not widely used nowadays and has not been applied for such kind of the problems yet. The end product introduced in this paper is a proof of concept, applicable for using Basic Test Sheets for asynchronous testing of real-time software systems.

The wide spread of systems with time constraints imposed on their response time in a web development caused an appearance of new technologies for asynchronous programming (i.e. node.js, python's tornado or twisted, ruby's event machine etc). This, together with increasing level of code reuse raised the speed of software development process. As a result, it raised the velocity of a new business requirements introduction. However, the processes of tests' definitions did not change a lot. It still has high entry level for people who define tests requiring them to have software development background. This fact necessitates direct involvement of software developers in the processes of test definition and editing. Test Sheets is a new approach developed by Software Engineering group of the University of Mannheim. It combines tool-free user experience of test definitions for business related staff via use of regular spreadsheets editors. As a result, it reduces involvement of technical staff to minimum.

This work describes the change project: development, adoption, conversion and use processes for an implementation of Test Sheets in figo GmbH. The research is based on works of Michael Zhivich and Robert Cunningham in the field of software testing, as well as papers of Robert Glass and Tsai, Fang and Bi in scope of real-time software systems testing. Description of asynchronous strategies analysed in this paper is based on General Theory of Reactivity, the research made by Chris Kowal and researches in field of reactive programming made by Erik Meijer, Cvonal Eliot and Mark S. Miller. The selection of the strategy for asynchronous event handling in node.js based on the performance measurements made by Gorgi Kosev. System architecture and design made with respect to the

principles of agile architecture described by Robert C. Marting and John Dolley. The implementation of the system made according to the Test Driven Development approach and guidances introduced by Robert C. Martin in his book "Clean Code: A Handbook of Agile Software Craftsmanship". The user experience collected by polling process based on After Scenario Questionnaire and Post Study System Usability Questionnaire designed by James R Lewis. The representation of system fit based on Task-Technology Fit model described by Dale Goodhue and Ronald Thompson.

The results of the research made by author within the scope of this paper are stated below. The guidance of code generation for asynchronous testing of real-time software was created (Section 10.4). The conventions (Section 9) for Test Sheet definitions created within the scope of this paper put limitations on a number of input/output parameters and reduce their type to javascript object. While enhancing the flexibility by introducing two types of comparison of actual and expected execution result. Transformation project dedicated to introduction of Basic Test Sheets into the business process of financial technology company described in Section 3. This paper introduces the comparison of Test Sheets with another widely used test definition approaches (Chapter 4). The definition of real-time software with respect to the modern state of art in web development is given in Chapter 5 which also describes specifics of a testing process for real-time software systems. Described architecture (Chapter 7), design and implementation (Chapter 8) together with principles and patterns this processes were based on. The measurements of performance and user experience were made are described in the chapters 11 and 12 representatively. The paper describes system fits and misfits together with benefits analysis of system's use (Chapter 13). Described limitations with guidance for future work in chapter (Chapter 14). The paper is closed with conclusions (Chapter 15).

2 figo GmbH

The German banking system is divided into 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 are represented by 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].

An introduction of the Payment Services Directive (PSD) and PSD2 by European Commission together with initiatives of the 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 the year 2017[9].

figo is a platform to access banking information and execute banking transactions in a regulated environment (PSD). figo connects and translates services (APIs) from 99% of German banks to a unified RESTful API for partners and third parties i.e. banks ans financial institutions, financial technology companies (FinTech)[11][12][32][17][35]. The list of figo's partners and customers together with their use cases can be found via following link: `http://figo.io/use_cases.html`.

In the scope of PSD2 the product of figo GmbH can be used in two different ways:

- Account Information Service Provider;
- Payment Initiation Service Provider.

figo GmbH is a 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"[17]. Currently, the API is fully functional in Germany, partly in Austria and England[11][12].

2.1 IT infrastructure: Banking Server

The high-level IT infrastructure of figo GmbH consists of two parts (Figure: 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 the connections to banks via three possible communication channels, their description provided below together with the basic motivation for each of them.
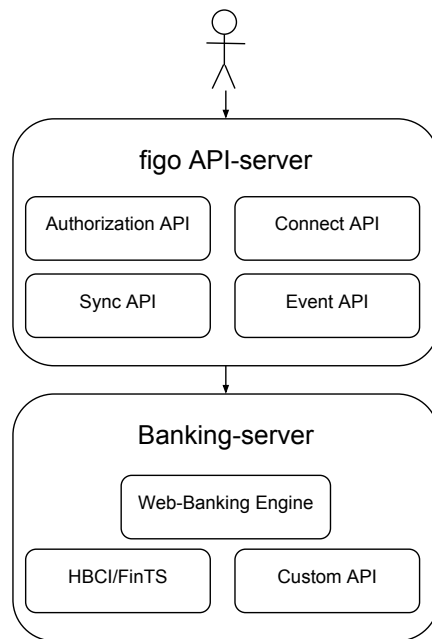
Figure 2.1: figo GmbH high level architecture

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 a client responsible for connection to custom APIs provided by banks. Some of them give full functionality while some only partial. In the same time all of them are an implementation of REST-API specification.

HBCI+/FinTS is responsible for connection to banks' interfaces via Home Banking Computer Interface (HBCI) client. It is an implementation of an *adapter OOP pattern* for the 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 such as the Bundesverband deutscher Banken e.V.*[13].

Web-Banking Engine is responsible for communication with banks which does not provide API nor 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. This is the most sensitive part from the developer's perspective, since every change to the bank's web page can lead to failure of the specific scripts. Tests defined with the use of Test Sheets are applied to testing of the scraping scripts against banking web-page.

3 Transformation project

This section describes the transformation project initiated by figo GmbH for the use of Test Sheets. **Enterprise System Transformation** - "is a hierarchic-sequential process involving all actions of individuals in an organization leveraging information technology to support the transition of an Enterprise System from its original O-I-T [Organization-Technology-Individual] configuration [...] to an (enhanced) configuration (Figure: 3)]"[50].
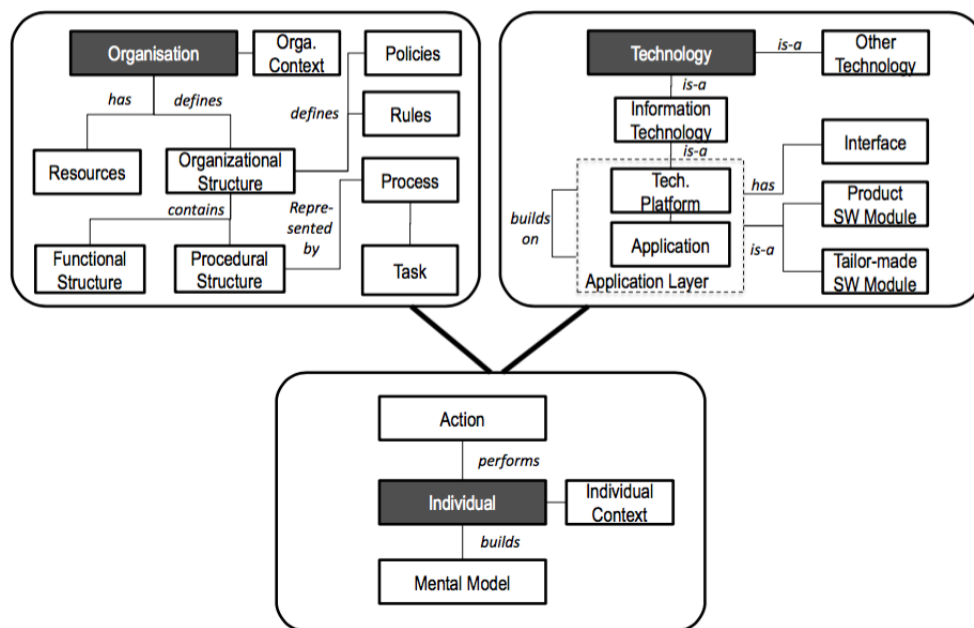
Figure 3.1: O-I-T model[50]

The transformation project includes four processes within itself:

Development focuses on the issue how information systems are developed. In contrast to software engineering, software development takes the socio-technical perspective on information systems[51].

Adoption   is a process defined by three concepts[52]

1. **Diffusion** refers to the breadth of use, e.g.the number of adopters in an organization;

2. **Infusion** describes how extensively the technology is used and the level of impact within the organization;

3. **Assimilation** addresses how extensively the new technology is used and how deeply the firm's use of the technology alters processes, structures, and organizational culture;

Convertion   is a process of realization of transformation projects with goal of successful system transformations [53].

Use   is a continues process closely coupled with conversion process. It covers not only the process of the system usage by the individuals but also the impact of it on the organization in context of short- and long-term benefits for individuals in the context of their performance and satisfaction [53].

3.1 Test Sheets project: Business perspective

Project definition: Introduction of a tool for verification and/or testing of product's functional fit to its requirements (i. e. detection of changes in banks web-pages, local software failures and/or errors) with minimal involvement of software developers into the process. Continuous project which requires software to be developed and supported by developers and can be executed on workstations of product/project managers.

Project uncertainties:
- Priority;
- Functional fit;
- Maintenance process;
- Functionality enhancement;
- Organizational culture fit.

Rationale for project creation and focus:

- High workload and time pressure in combination with limited technical resources leads to less time available on manual system monitoring. Non developers were not capable of testing the functionality;

- Output of the tests results will help in a time estimation for errors fixing.

Underlying assumptions: Shift of responsibility for functionality on managers will improve system maintenance process and customers' user experience.

Goal: Routinization of product verification/testing process with respect to its business and functionality requirements by use of Test Sheets. Tests defined with Test Sheets must be executed on timely basis for immediate detection of critical changes in web pages of clients for Internet banking.

Success:

- Goals are met;

- The system is fully routinized within the business processes;

- The time for bugs allocation and fix is reduced;

- The number of affected customers is minimized.

3.2 Test Sheets project: Technical perspective

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.

Software developers must be notified about script failure as soon as it was detected to improve the response time.

Scraping scripts are communicating with web-pages in a real-time and asynchronous fashion.

The language of implementation must be javascript for low entry level of developers in the 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.

3.3 Test Sheet's project plan

This high level plan was developed and used by the author to align the project state, to perform time estimations and to provide transparency regarding the project state for company's management staff:

1. Test Sheets concept introduction to the target customers;

2. Requirements analysis;

3. Mapping requirements to the concept of Test Sheets;

4. Definition of the development process;

5. System design;

6. Development:

   a) ALPHA1 version without support of references and linear tests execution;

   b) ALPHA2 version with references support and linear tests execution;

   c) BETA version with optimization for real-time testing;

   d) Reporting mechanism;

7. Demonstration;

8. Performance measurements;

9. Workshop on system installation and usage;

10. User experience collection;

11. Routinization.

4 Software testing

Software testing is an important technique to evaluate and assess software's quality and reliability, it provides a possibility to determine whether the development of product conform the requirements. At least 30% of the price of a software project is the cost of software testing [40]. The 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 economic 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 the USA lead to losses of $6.5 million per hour in financial services and $2.4 million per hour in credit card sales applications[39].
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[26].
In general, according to report made by US National Institute of Standards and Technology (NIST) in 2009 the estimated economy loses were $60 billion annually as associated with developing and distributing software patches and re-installing systems that have been involved, together with losses in productivity due to errors and malware infections[26].

The human causalities caused by software misbehavior can rich dramatic scales.
The case of Toyota unintended acceleration caused by software defect in 2009 - 2011 led to 89 deaths and 57 injures. Toyota was fined $1.2 Billion for concealing safety defects[6].

The Patriot, surface-to-air missile, system failed to track and intercept an incoming Scud missile on 25 of February 1991. As the result 28 soldiers were killed and around 100 were injured[26].

A Therac-25, linear accelerator, which failure caused to 6 know cases where cancer patients received deadly radiation overdoses during their treatment between June 1985 and January 1987. The dosages were 100 times exceeding typically used for treatment[26][46].

4.1 Existing approaches for test definitions

In most of the cases developers are using testing frameworks or internal Domain Specific Languages (DSL) which requires the use of formal programming languages. This requires knowledge of languages and understanding of the basic programming concepts from everyone who is involved in the creation of a test, reading its result or updating/deleting the test. The brief overview of different testing approaches and their analysis with respect to modern business requirements is provided below.

4.1.1 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[55][37]. The general simplicity and lightweight made them as a popular tool for Test Driven Development[37].

xUnit basic features implemented by all members of the family provide functionality to perform following tasks[55]:

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 xUnit's 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.)[38].

Definition of tests with formal general programming language from one side makes tests definition and execution fast but at the same time it rises the entry level required for a creation of tests for people without developer background.

4.1.2 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 a feedback loop between them. Fit automatically checks HTML pages against actual program[14].

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[15].

Programmers use a **ColumnFixture** to map the columns in the table to variables and methods in the fixture. The first columns provide correspondence to variables in the fixture. The last column has the expected result and corresponds to the method in the fixture[15].

Different implementations 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[14] requiring developer to write only **fixture**, the middleware between tests and code. At the same time, it provides media between operations over tests and the results of their execution for people without development experience.

4.1.3 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 more then sixty different languages[8].

Feature files written in Gherkin consist 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 at the same time software development experience required for writing code with step definitions.

4.2 Test Sheets

Test Sheets is a representation for tests 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[28].

Test Sheets approach uses a usual spreadsheet for a 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, the result of tests execution is provided in the same table with coloring and actual return are separated by "\" symbol in case of failing test[29].

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 a tool free user experience for tests definitions. Which can be done by a user without prior development experience with application of any spreadsheet editor.

4.2.1 Basic Test Sheets

A Basic Test Sheet is a type of Test Sheets with sequential tests execution without parametrized values, but with the 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;

The values in columns cells 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;

The description provided above is shown in a summarized example from the web site of Chair of Software Engineering of University of Mannheim (Figure: 4.2.1).

Figure 4.1: Basic Test Sheet

The software developed within the scope of this research is a proof of concept for the 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/`.

5 Real-Time Software and its testing

5.1 Real-Time Software

Donald Gillies [27] 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 [45] defines this term as: *"Real-time software is 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 define Real-Time software as a combination of this two definitions and covers both logical and time correctness as well as an interaction with external systems controlled by the ongoing process. *Real-Time Software is a type of software in which correctness of computations based on both internal and external inputs is valid if and only if the result of the computation is logically correct and was obtained within the defined period of time.*

The Web-Banking engine of figo GmbH matches this definition due to the following facts. First, it performs communication with external systems (API customers input and Web Banking HTML pages). Next, its result depends on 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 considered to be the failure) as well as logical correctness depended on the ability of the script to perform necessary actions for a fulfillment of a requested task.

5.2 Testing Real-Time software

Tsai, Fang and Bi[57] 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 the 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 an internal criteria but also by real world processes and their timing constraints.

The general testing and debugging strategy (Figure: 5.2) described by Tsai, Fang and Bi [57] consists of four steps.
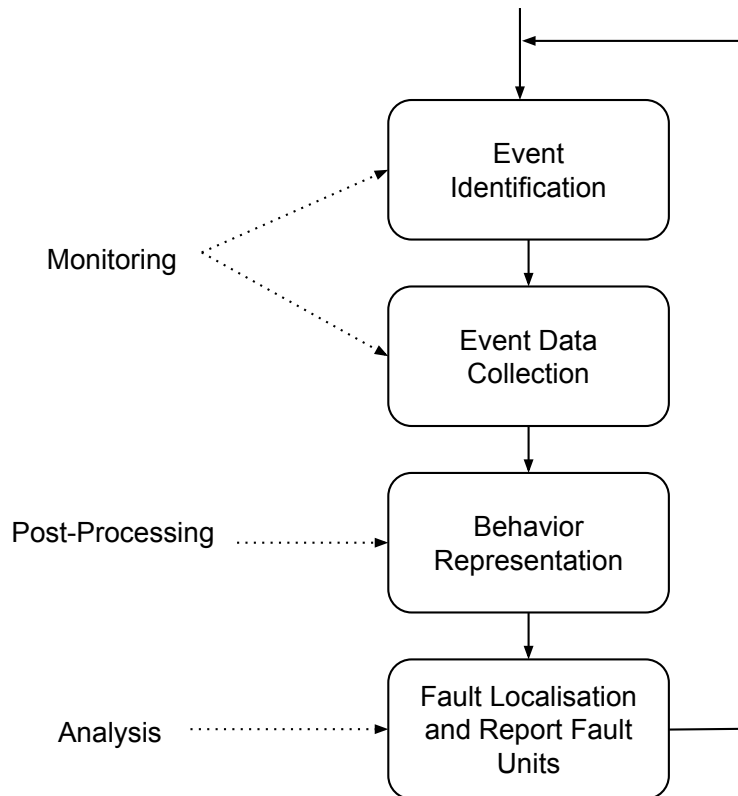
Figure 5.1: General Testing and Debugging Strategy[57]

At the first step, *Event Identification* 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 *Event Data Collection*, the execution data, containing the event key values identified in the first step, is collected using the real-time monitoring system. The first and second

steps constitute the monitoring phase. After the data is collected, it is processed in an off-line mode to construct logical views to represent the program's execution behavior at the third step, *Behavior Representation*. At the *fourth* step, *Fault Localization and Report* analysis algorithms will be employed to identify fault units and report them to the user. Finally, the user can use this report to decide next monitoring and debugging cycle for a lower level abstraction.

5.3 Web scraping

Web scraping is a method for extracting information from web pages[33]. 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 a useful technique for real-time data extraction from human readable web-pages. W3Council [31] defines DOM 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."

5.3.1 Web scraping tools

PhantomJS [25] `http://www.phantomjs.org` is a Web Browser Engine without graphical user interface (headless WebKit) scriptable with JavaScript.

Use Cases:

- Headless web testing (without the browser).

- Page automation. Access and manipulate web pages with the standard DOM API, or with usual libraries like jQuery.

- Screen capture. Programmatically capture web contents, including CSS, SVG and Canvas.

- Network monitoring. Automate performance analysis, track page loading and export as standard HAR format.

Features:

- Multiplatform, available on major operating systems: Windows, Mac OS X, Linux, and other Unixes.

- Native implementation of web standards: DOM, CSS, JavaScript, Canvas, and SVG.

- Pure headless.

CasperJS   is an open source navigation scripting and 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 and syntactic sugar for doing common tasks for interaction with DOM of the web page[7]:

- defining and ordering browsing navigation steps

- filling and submitting forms

- clicking and 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

The use of this tools 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 a web-page or its part defined as a 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.

6 Asynchronous programming strategies and their performance

6.1 Asynchronous programming

Asynchronous programming is the programming model in which operations that must be executed are interleaved with one another within the single control thread. An analogy to it can be package multiplexing in a Computer Networks.

In contrast multi-threaded systems are synchronous and allow execution of one task per unit of time blocking execution of other tasks until programmer will perform explicit control delegation to another 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 of one another;

6.1.1 node.js

node.js is an asynchronous event-driven framework designed to build scalable network applications. node.js' design is similar to and influenced by systems such like Ruby's Event Machine or Python's Twisted with the difference that it presents the event loop as a language construct but not as a library[1].

node.js as a framework has powerful ecosystem - npm which consists of three parts. *npm* - package manager for node.js, *npm Registry* - public collection of packages of open-source code, *npm command line client* which allows developers to install and publish those packages. The diagram (Figure: 6.1.1) states the

Module Counts
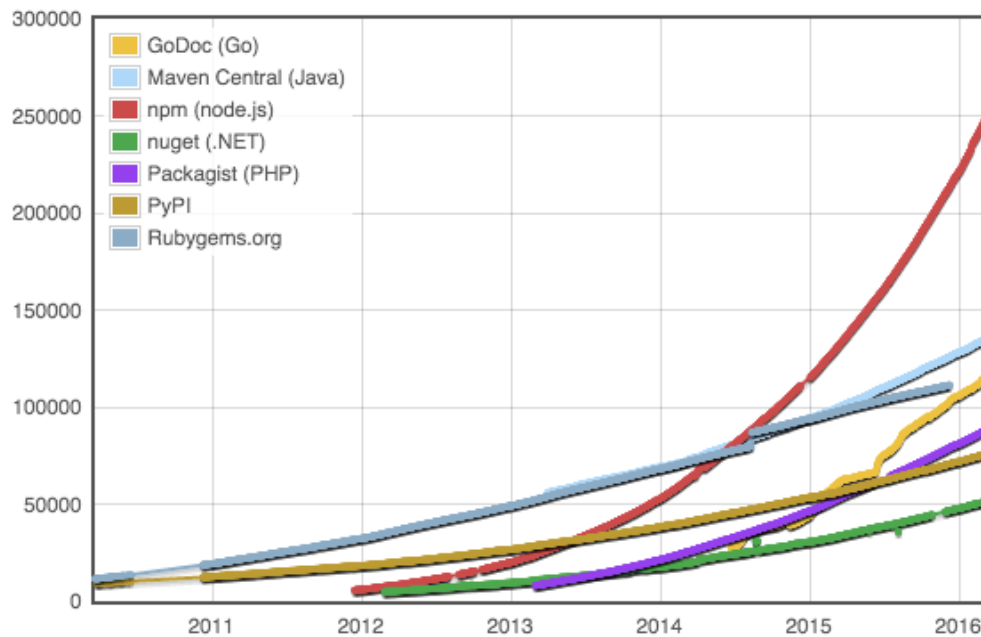
Figure 6.1: npm comparison with other package managers[23]

comparison of npm with other package, for more up to date information please visit `http://www.modulecounts.com/`.

At 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 by the failure during the build process for thousands of projects[42]. Moreover, there are three factors explained by Sam Saccone regarding this topic [56] [43]. All of them make npm a root source of hackers attack since all life-cycle scripts executed by installed packages are executed with same privilege level as a user invoked their installation. The *first* reason is caused by the usage of SemVer for the version controlling which does not lock dependencies to a specific version The *second* reason is the lack of automatic npm user log-out. npm can run arbitrary scripts on install it means that any user who is currently logged in and types npm install allows any module to execute arbitrary publish commands. The *third* reason is dependent on the fact that a singular npm registry is used by large majority of the node.js ecosystem on a daily basis.

General suggestions to overcome problems for project developers and maintainers are following:

- disable life-cycle scripts during update/install process;

- lock down all dependencies versions either manually or using shrinkwrap package `https://docs.npmjs.com/cli/shrinkwrap`

- store dependencies locally and use them as a part of source code being deployed.

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[20].

The high-level architecture of node.js consists from the following parts (Figure: 6.1.1):

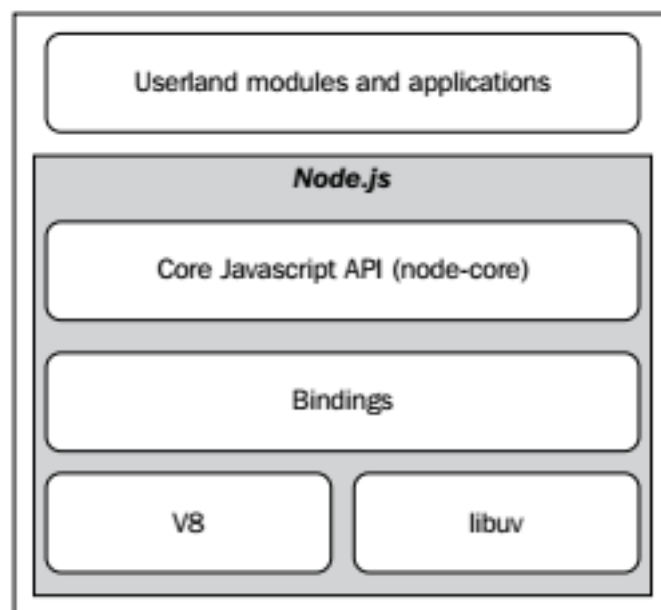Figure 6.2: Node.js architecture [41]

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.[41]

Non blocking I/O provided by libuv[1][41]. Which is "a multi-platform support
library with a focus on asynchronous I/O. "[34] 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"[41]

6.1.2 Event handling

An 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 an event is emitted all the functions attached
to that specific event are called synchronously[10]. Node.js event handler is an
implementation of *reactor pattern*. The illustration of process life-cycle is shown
on the figure 6.1.2:

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 complete, 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 listener, which is part of the application code, will give back the con-
   trol to the Event Loop when its execution completes (5a). However, new
   asynchronous operations might be requested during the execution of the
   handler (5b), causing new operations to be inserted in the Event Demulti-
   plexer, 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.
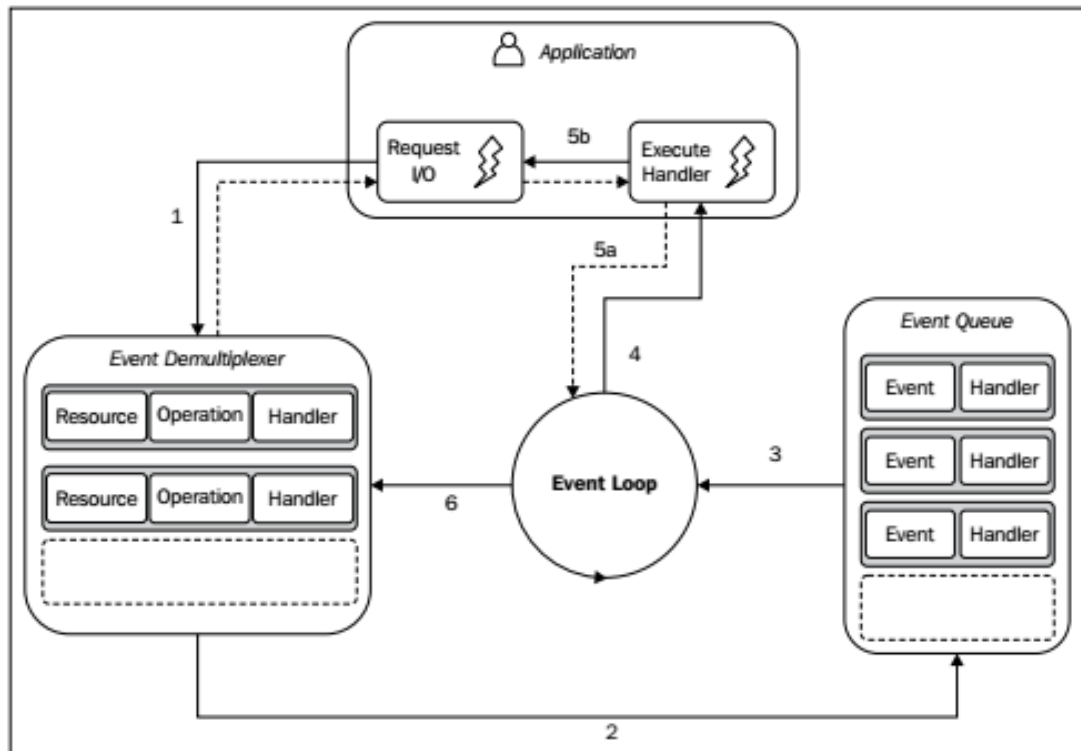
Figure 6.3: Node.js event handling system [41]

6.2 Asynchronous handling strategies

6.2.1 Declarative

The presence of functions as the first class citizens in javascript allows direct usage of functions for handling asynchronous program behavior. *Callbacks* are default handlers for the reactor pattern described before. They are similar to *visitor pattern* in OOP or continuation-passing-style from Functional Programming. They represent an operation to be performed on the elements of an object structure and they let define a new operation without introducing any changes to the definition of the object. By the 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*[16]. This occurs in case of inconsistent function behavior when under some hidden conditions a function performs a synchronous action but some under other - asynchronous.

6.2.2 Imperative

The further explanation is based on the *concept of duality* (Figure 6.2.2) showed by Erik Meijer `https://channel9.msdn.com/Events/Lang-NEXT/Lang-NEXT-2014/Keynote-Duality` and used by Kris Koval [21] in his *General Theory of Reactivity* `https://github.com/kriskowal/gtor/`.
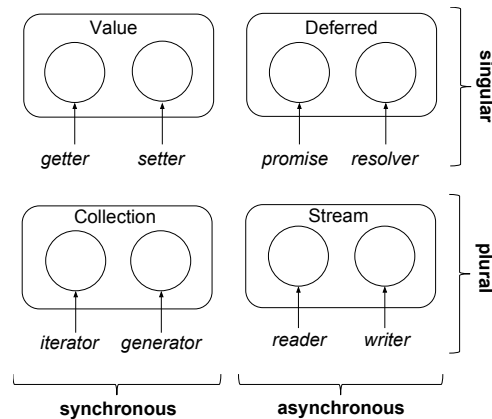
Figure 6.4: Duality Matrix [21]

The explanation of asynchronous patterns in this chapter will be made as two-dimensional mapping. The first dimension is a time where things can be *synchronous* and *asynchronous*. Another dimension of space where mapping is done between *singular* and *plural*.

Synchronous: **Value** is a singular unit data. Its duals are *getter (pull)* and *setter (push)*. Setter accepts a 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 the same logic to their analogs for further entities. A **Collection** is a plural form of the value. The duals of a collection are *iterator (pull)* and *generator (push)*. An *iterator*

as a plural analog of getter, it accepts nothing and returns the element from collection. A *generator* is a plural analog of setter it accepts element to be added to collection and returns nothing.

Asynchronous: Deferred is an 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 the combination of read and write.

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

6.3 Performance

Following measures are taken from the article "Analysis of generators and other async patterns in node" by Gorgi Kosev [36] [2] . There were no hardware charac-

teristics provided for the experiment execution environment except the following description[36]: "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) [36]

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

Table 6.1: Performance comparison of patterns for asynchronous information flow [36][2]

According to the author of measurements [36] "[2] "The original and flattened solutions are the fastest, as they use vanilla callbacks.

Note that this table have 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 since it is low level implementation on C++. For more details please visit `http://bluebirdjs.com/docs/getting-started.html`.

7 Architecture

The system consist of three piped object streams (Figure: 7). Stream piping implements the idea of pipe "|" in Unix systems invented by Douglas Mcllroy [30]. It enables the output of one program to be connected to the input of the next program. *Object stream* (stream in an object mode) is a stream in which a data is treated as a sequence of discrete javascript objects.

Pipe method of streams provide developers with an 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.

The piping of object streams allows to perform parallel executions which can be beneficial from the performance perspective. Following streams represents the high level architecture of the system. The scope of their responsibilities provided below:

1. **Read Stream** accepts directory address and *pulls* content of all spread sheet files together with their metadata from this directory including nested directories. From this stream perspective the *Transform Stream* will be referred as an *up stream*;

2. **Transform Stream** accepts the data object with the file name, content, metadata from upstream, creates TestSheet schema and generates content of javascript file implementing *interpreter pattern*. From this stream per-
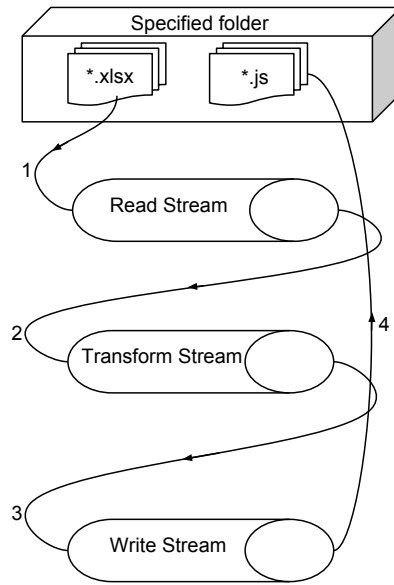
Figure 7.1: Information flow

spective the *Read Stream* will be referred as an *down stream*, and *Write Stream* as an *up stream*;

3. ***Write Stream*** *pulls* data from upstream. Then is performs an attempt to read the representative .js file from the specified folder. If the file exists and its last update date is older then for spreadsheet file then the next step will be skipped. If the file does not exist or its the last update date is earlier then the last update date of spreadsheet file then it creates/overwrites javascript file. From this stream perspective the *Transform Stream* will be referred as an *down stream*;

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 in case if a source xlsx file was not updated.

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/

Each directory unifies node modules for implementation of representative logic. Every folder has *index.js* file which is the public interface file of the module. The default behavior of the *require* method from node.js is to require the *index.js* file from folder if path to it is passed as an input parameter, and file if an input parameter is a path to the file.

The entry point of the system ./index.js looks as following (Listing: 7.1):

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

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

Listing 7.1: index.js

The program invoked from the command line interface accepts one parameter - path to the folder with xlsx files which stores Test Sheets defined by users. The *argv* property of *process* is an array of parameters with which the node module was called from a command line interface. The first parameter is a module name. All next parameters are the arguments passed to this module.

8 Design Principles

The implementation of each part of the system was made with application of Test Driven Development.The testing framework is *mocha* https://mochajs.org/. For the creation of doublers (mocks, stubs and spies) for object and method was used *sinon.js* http://sinonjs.org/.

Application of Test Driven Development principles provide better control over the system on a code level but not on design. In the same time Robert Martin [54] defined symptoms of not agile design as following:

- 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 [44] and Rober Maritn [54] suggested the following 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 the duplication of the code.

- The Single-Responsibility Principle - A class should has the only one reason to change from the business perspective;

- The Open/Closed Principle - Software entities (classes, modules, functions, etc.) should be open for the 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 are suggestional and can not be strictly followed due to their mutually exclusive nature.

9 Conventions

Following conventions were developed by author with respect to the requirements defined by figo GmbH and based on the definition of the Test Sheets. This conventions must be followed by Test Sheet for proper work of developed system:

- Number of columns within one Test Sheet should not exceed 26 columns (from A to Z);

    - **A1** cell - description of the test case;

    - **A2** cell - module under test (folder with index.js or file with js extension);

    - **A3..n** cells - name of the class/object under the test;

    - **B3..n** cells - name of the method from representative class/object under the test;

    - **C2..n** cells to **Invocation Column** - input parameters for representative method under the test;

- Invocation delimiters must be allocated within the single column (aligned to the longest row);

    - **Invocation Column** - the column for separation of input values from expected output value(s) filled with | (pipe)(for comparison by scheme and data types) || (two pipes)(for deep comparison - by scheme, data types and values) as a cells values until the last line which includes objects under tests;

- **Expected Return** - column must be next after invocation line.

- All functions under test should be asynchronous and accept callback as a last input parameter with standard signature for handling emitted event;

- References to the columns with expected return values will accept as value the value obtained from the method execution;

- References must be defined only to cells in one of the previous rows;

- Files extensions must be xlsx.

10 Implementation

For the description of data structures processed by the system, we will use following module (Listing: 10.15) which implements a stack with asynchronous methods (respond time for each method is delayed 10 milliseconds). Usage of the *setTimeout()* method guarantees its asynchronism. While the fact that the timeframe for the response is hard coded into scraping scripts ensures that this method is valid for usage as a proof of concept within the current research topic.

```javascript
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 10.1: stack.js

The Test Sheet for test coverage of this module looks as following (Figure: 10.1). Note that this test coverage was made for the purpose to show handling of asynchronous testing of real-time software but not to cover the stack module. There are both types of comparison in invocation lines (Column **D**). The red arrows indicate the 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 10.1: Test Sheet coverage for stack.js

For the reading of the content from the spread sheet and transforming obtained information into javascript code with further writing of it into a js file the system uses following dependencies upon external npm packages:

- Development dependencies:
  - "fs-readdir": "0.0.3", (https://www.npmjs.com/package/fs-readdir);
  - "through2": "2.0.0", (https://www.npmjs.com/package/through2);
  - "xlsx": "0.8.0", (https://www.npmjs.com/package/xlsx);
  - "multipipe": "0.3.1", (https://www.npmjs.com/package/multipipe)
- Testing dependencies:
  - "mocha": "2.3.4", (https://www.npmjs.com/package/mocha)
  - "sinon": "1.17.2", (https://www.npmjs.com/package/sinon)

10.1 Read Stream

This part of the system is an implementation of combining streams pattern using *multipipe* module. From the internal view, it consists of two streams. The first stream is implemented with the use **getFilesStream** function from **fs-readdir** module. It creates readable stream for reading content of directory including nested directories and returns the array with absolute paths to them. Second stream is **getDataStream** the stream in an object mode. For every file path, it accepts from the down stream, it checks its extensions and if it is equal to *.xlsx* this stream reads the content of files. For this purpose it uses **xlsx** module. While for reading files' metadata the internal node.js module **fs** is used. Next, this stream pushes obtained data to the up stream in a data object with file name, file meta data and its content (Listing: 10.2).

```js
function getFilesStream(dirPath) {
  return fsReaddir(dirPath)
    .on('error', function(obj) {
      console.error(obj);
    })
    .on('finish', function(obj) {});
};


var getDataStream = through.obj(function(files, enc, callback) {
  for (var file of files) {
    if (path.extname(file) === '.xlsx') {
      var sheet = xlsx.readFile(file).Sheets.Sheet1;
      var meta = fs.statSync(file);

      this.push({ fileName: file, meta: meta, sheet: sheet });
    };
  };

  callback();
});

module.exports = function(dir) {
  return multipipe(getFilesStream(dir), getDataStream);
};
```

Listing 10.2: read_stream.js

This module exports single function which accepts a single string parameter - path to the directory with Test Sheet files. This function return single stream (two piped streams), an array of deferred values which will be pushed to the upstream. Each of deferred value is a javascript object with following structure (Listing 10.3)

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

Listing 10.3: Read Stream Output / Transform Stream Input JSON

10.1.1 Test Coverage

Following test cases show step every incrementally performed during the development process. All calls to systems I/O (e. g. read file meta data, read file) are performed 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 or functions with predefined returned values.

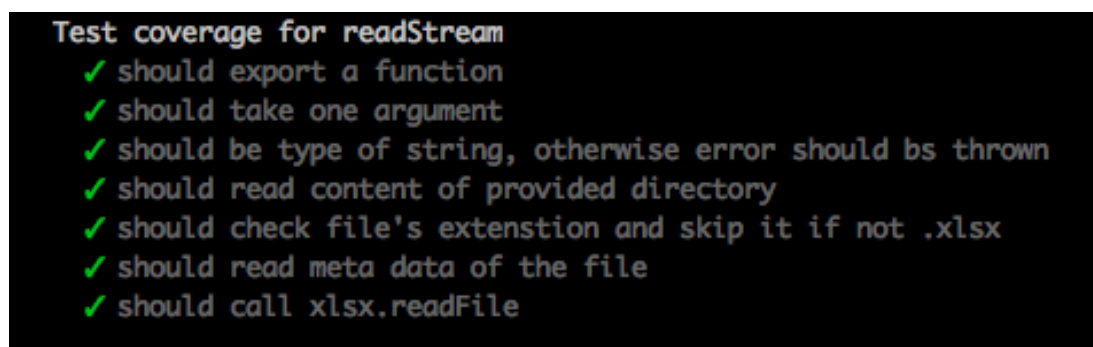The result of tests execution for this module is following (Fig.: 10.2):

Figure 10.2: Test coverage for read_stream.js

10.1.2 Correspondence to the design principles

- Closing - the part of the code responsible for the reading content of the file with the concrete extension and the concrete structure is encapsulated with

getDataStream function

- Code to an Interface - no references to any classes;

- Do not repeat yourself - no code duplication;

- The Single-Responsibility Principle -

- The Open/Closed Principle - functionality of the library can be extended via adding new pipes with application of different piping patterns in a single place;

- The Liskov Substitution Principle - object composition is used over inheritance (function wrapped in to pipe object);

- The Dependency-Inversion Principle - communication with other modules is done via standard interface.

  - 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 - composition over inheritance and usage of standard interfaces;

- Principles of Least Knowledge - communication done only with *xlsx* files;

- The Principle of Loose Coupling - communication is done via standard file system and standard stream interfaces.

10.2 Transform Stream

This part of the system performs translation of the Test Sheet into executable javascript code. From the perspective of this module the translation performed in two stages (Listing: 10.4)

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

function transform(data, enc, callback) {
  var res = {
    fileName: data.fileName,
```

```
         meta:  data.meta,
         content:  template.applyTemplate(
10             data.sheet,
             scheme(data.sheet),
             data.fileName),
      };

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

Listing 10.4: transform_stream.js

The **first stage** is a creation of *scheme* object of the Test Sheet. The **second
stage** is an application of a template to the content of Test Sheet and the ex-
ecution schema which has been created on a previous stage. The result of this
transformation is pulled by the upstream as a *content* property of an object to-
gether with *meta data* and *fileName* obtained from the downstream (Listing:
10.5).

```
   {
      fileName:  <String>,
      meta:  <Object>,
      contentent:  <String>
5  }
```

Listing 10.5: Transform Stream Output / Write Stream Input JSON

10.2.1 Test Coverage

Following test cases can show every step incrementally performed during the
development process. Including invocations of helper methods from *scheme* and
template modules.

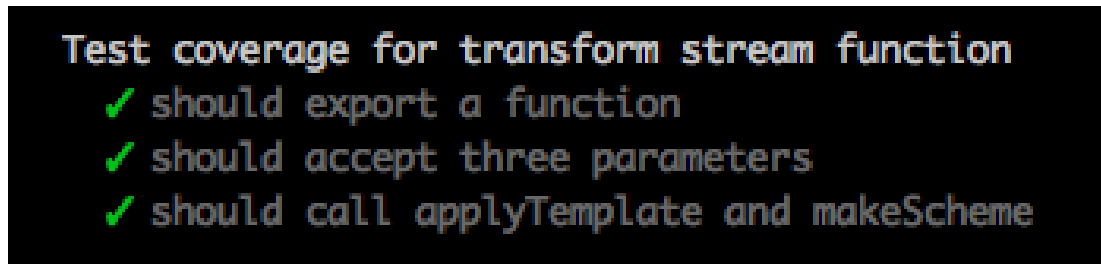The result of tests execution for this module is following (Fig.: 10.3):

Figure 10.3: Test coverage for transform_stream.js

10.2.2 Correspondence to the design principles

- Closing - the creation of data structure to be transformed to the next stream is created in a single function;

- Code to an Interface - this module does not refer to any class.

- Do not repeat yourself - no code duplication;

- The Single-Responsibility Principle - can be changed only due to the change of Test Sheet type (another type of scheme);

- 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 - object composition over inheritance (function wrapped in to pipe object);

- The Dependency-Inversion Principle -

  - Higher level module *index.js* does not depend on implementation current library;

  - Current library depends on the input object provided by the previous stream;

- The Interface Segregation Principle - does not depend on methods since it only creates data object to be passed to the next stream;

- Principles of Least Knowledge - direct communication done only with helper functions for schema creation and content generation.

- The Principle of Loose Coupling - communication is done via the standard stream interface

10.3 Scheme

This part of the system creates a scheme object from the Test Sheet, the only parameter accepted by the main function of this module is a Test Sheet object (the content of xlsx file in a JSON format).

The **first stage** is creation of the object with following properties:

1. description;

2. moduleUnderTest;

3. objectsUnderTest;

4. methodsUnderTest;

5. inputs;

6. invocations;

7. outputs;

The values of each property except *description* and *moduleUnderTest* are arrays of strings with values which represent cell addresses for representative property of the Test Sheet. For the first two properties the values are strings with data from the cells *A1* and *A2* respectively(Listing: 10.6).

```
{
  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 10.6: Result of the first stage of Test Sheet scheme creation

The **sectod stage** is a pivot transformation of the scheme provided from the 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 next of them are objects which consist of following Test Sheet properties: *objectUnderTest, methodUnderTest, inputs, outputs, invocations.* With values as single element arrays with coordinate of respective cell (Listing 10.7).

```
{
  "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"],
40          "methodsUnderTest":["B10"],
            "inputs":[],
            "outputs":["E10"],
            "invocations":["D10"]},
      "11":{"objectsUnderTest":["A11"],
45          "methodsUnderTest":["B11"],
            "inputs":["C11"],
            "outputs":["E11"],
            "invocations":["D11"]}
   }
```

Listing 10.7: Result of the scheme creation for Test Sheet

10.3.1 Test Coverage

Following test cases can show every step incrementally performed during the development process. Since the object obtained from the reading of xlsx file is passed from one part of the system to another (*Object pool OOP pattern*) this module does not perform any I/O calls. There are getter function for each type of cell in a Test Sheet as well as pivot function called *transformScheme* and helper for it called *getRowFromField*.

The result of test execution (Figure 10.4):

Figure 10.4: Test coverage for scheme.js

10.3.2 Correspondence to the design principles

- Closing - comparison type is closed within the *getInvocationCells* function, all elements of Test Sheet are obtained via separate getter functions;

- Code to an Interface - all methods have the same minimal abstraction layer

    for object which represents the content of Test Sheet. All keys of the object should be named as a two dimensional addresses of the cells (i. e. *A1*, *D7* etc);

- Do not repeat yourself - no code repetition.

- The Single-Responsibility Principle - *transformScheme* function should be changed in case of introduction of the non-linear Test Sheets, since this type has behavior definition in rows under test steps;

- The Open/Closed Principle - violated in a *transformScheme* function;

- 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;

- The Principle of Loose Coupling - exports single function;

10.4 Execution Order

This part of the system is responsible for the definition of an order for the test steps execution. It is called in the main entry point of the scheme module and applied to the pivoted scheme described above and the sheet object the scheme relates to. In a general case, the execution order of tests for a Basic Test Sheet is defined by the order of test steps. All Test Sheets support references it 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 the next function can invoked. Moreover, *real-time* systems have time constraints on their execution, but do not provide an exact time within it will be completed. Further, due to the requirements for the generated tests proposed by figo GmbH test execution should be fast since it will be performed in a timely fashion. Combination of all this factors forced to define moment of each test step execution with respect to the source of their input parameters.

For the definition of test steps execution order the decision was taken is to separate test steps in two types. *The first* type covers test steps which behavior does not make influence to the 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 the next test steps or is 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 the Test Sheet and *scheme* - the object generated by the main function from *scheme.js* file. Walking through the scheme elements starting with the third (skipping service rows) performs the check for each cell in a sheet object for determining does behavior of test step define inputs of any next test step. In this case such a 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 the global (in a function scope) *chains* array. All other rows will be pushed to the *linear array* (Listing: 10.8).

```
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 10.8: Generation of tree structure as a list of childnodes

For the further creation of test steps execution order the list of child nodes should
be merged into nested list for child nodes which has other child nodes. In terms
of test steps this means that the result one test step can define the behavior of
few next steps either directly or via some intermediate test steps. The merging
process is performed by the *transform* function (Listing 10.9). 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
checks the values of childes (nested arrays) for each of them being a parent for
other child nodes (is it present as a first element in next arrays). In such case it
takes list of its child nodes and creates a two dimensional array which replaces the
the parent element in a list of childes with removing parent entry together with its
child nodes from the current position. This operation of replacement performed
recursively through all nested arrays using *isIn* function which determines if given
element belongs to the provided array including nested arrays.

```
function transform(arr){
   for (var i = arr.length - 1; i >= 0 ; i--) {
      for (var j = i - 1; j >= 0; j--) {
         if (isIn(arr[i][0], arr[j][1])) {
            var index = arr[j][1].indexOf(arr[i][0])
            arr[j][1].push(arr[i][1])
            arr.splice(i, 1)
            break
         } else if (arr[i][0] === arr[j][0]) { // merge same level
               nodes
            arr[j] = [arr[j][0], merge(arr[j][1], arr[i][1])]
            break
         } else if(isIn(arr[j][1][0], arr[i])){
            break
         } else {
         }
      }
```

```
  }
  return arr
}
```

Listing 10.9: Generation of nested data structure

The final step is the concatenation of arrays from the *first* and the *second* test steps types. In this example, rows *one* and *two* belong to the first type as a service rows which do not define any test steps together with rows *three* and *four*. Note that the input parameter for the test step defined in a row number four is referenced by the expected output of the test step that was 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 the execution starts without any logic violation. At the same time result of test step *five* is an input parameter for the steps *six*, *seven* and *ten* this in turn defines an input for the test step *eleven* (Listing: 10.10).
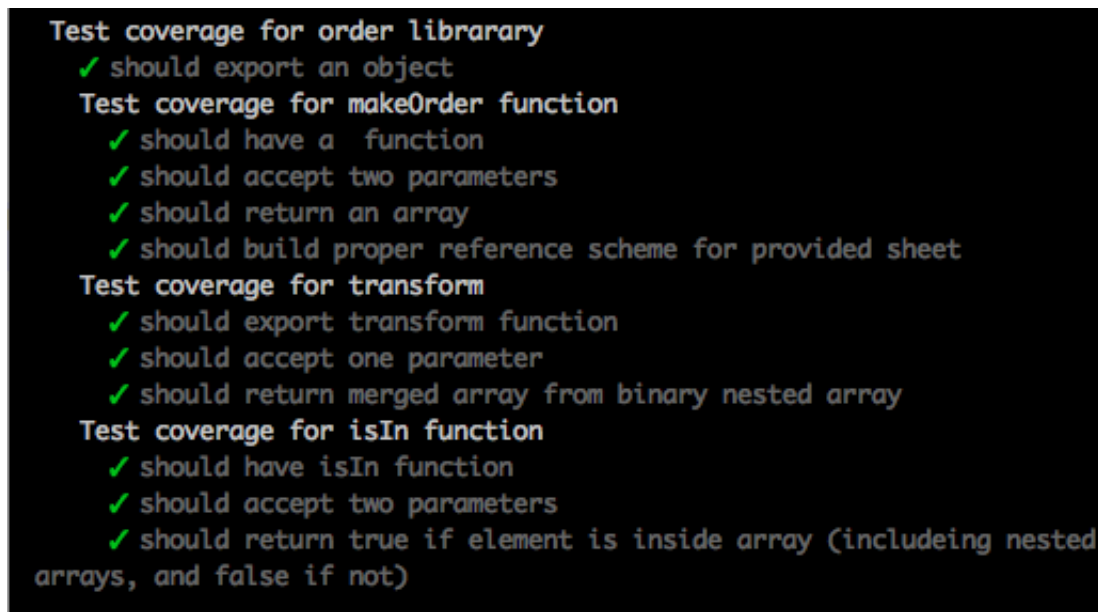
```
[
   1 ,
   2 ,
   3 ,
   4 ,
   [ 5 ,[ 6 ,7 ,[ 1 0 ,[ 1 1 ]]]] ,
   [ 8 ,[ 9 ]]
]
```

Listing 10.10: Execution order

10.4.1 Test coverage

The following test cases can show every step incrementally performed during the development process. Here you also can see application of Object pool pattern which helped to get rid of the call to I/O for the process of reading content from the .xlsx file. The result of test execution (Figure 10.5):

Figure 10.5: Test coverage for order.js

10.4.2 Correspondence to the 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 - can be changed only because of the introduction of new types of the Test Sheets;

- 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;

- The Principle of Loose Coupling - exports a single function.

10.5 Execution Scheme

This step performs the recursive replacement of elements in an *execution order* array with the elements from the *scheme* object. The function (Listing: 10.11) accepts two arguments: *scheme* object and *order* array. If an element is of an input array is an array it then 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 10.11: Application of execution order to the scheme

10.5.1 Test Coverage

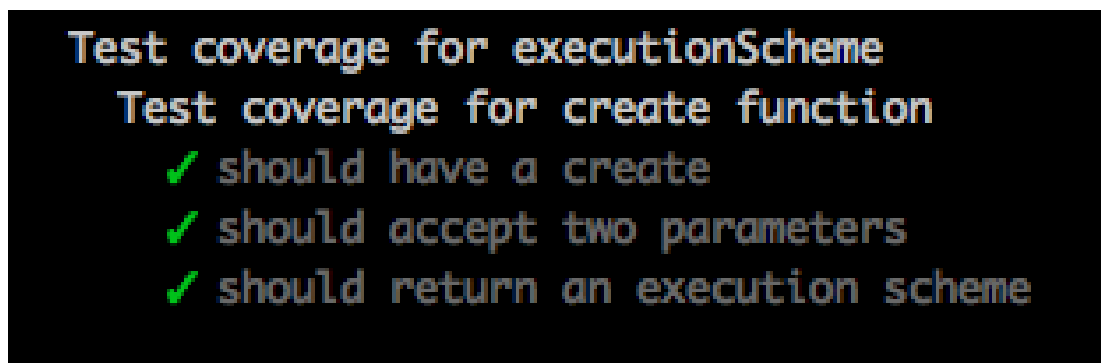Following test cases can show each step incrementally performed during the development process (Figure: 10.6).

Figure 10.6: Test coverage for creation of execution scheme

10.5.2 Correspondence to the 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 add/change the 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;

- The Principle of Loose Coupling - exports a single function.

10.6 Template

The translation to executable javascript file is made by the application of 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. This function also make indentation and formating of generated file content. The creation takes part in a four stages.

The **first stage** is adding a description from the first element of the scheme object. It is 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 the variables are taken from the cells with respective addresses.

The **fourth stage** is adding calls to the methods (Listing: **??**). This stage is the most important part of this module. Since there are two types of execution orders adding function calls is performed for two cases described below.

The ***first case*** is adding calls for independent test steps. This is made by iterative call of the *makeCall* function with appending closing brackets to the end of the result string during each of iterations. This function simply performs mapping between *sheet* object (content of xlsx file), *scheme* entry (test step) and string with the content for executable javascript file which represents each individual test step from the Test Sheet (Listing: 10.12).

```javascript
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 10.12: Function makeCall from template module

Note that functions in the generated code are not called directly but via the *call* method of the function's prototype. This method provides an opportunity to call the function from the different context (environment). In the 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 the creation of child process. The environment for such calls should be passed as an *object under test*.

The **second case** is adding calls for interdependent test steps. For every element of the *nestedCalls* array it calls the *makeNestedCall* function. Which calls *makeCall* function (Listing: 10.12) and if the next element is an array then this function performs recursive call with an appending of the closing brackets to the result of this call. In case if next element is not an array the it appends closing brackets and performs next iteration (Listing: 10.13).

```
function makeNestedCall(sheet, nestedCalls, fileName, str, indent){
  for (var i = 0; i < nestedCalls.length; i++) {
    str += makeCall(sheet, nestedCalls[i], 0, '', fileName)
    if(Array.isArray(nestedCalls[i+1])){
      return str += makeNestedCall(sheet, nestedCalls[i+1], fileName
          , '', 0) + '\n});\n';
    } else {
      str += '\n});\n'
    }
  }
  return str;
}
```

Listing 10.13: Function makeNestedCall from template module

The general case signature for function call generated from this template is following (Listing 10.14):

```
<module under test>.<method under test>.call(<module under test>, [<
    input paramteres>],function(err, <actual output>){
  if(err)  return(err, null);
  makeComparisonAndWriteResult(<expected output>, <actual output>, <
      comparison type>, '<module under test> <method under test>', '<
      path to xlsx file>', '<actual output>');
  <expected output> = <actual output>;

});
```

Listing 10.14: Signature for generated function call

10.6.1 Test Coverage

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

The result of test execution (Figure: **??**):

Figure 10.7: Test coverage for template.js

10.6.2 Correspondence to the 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, new functionality can be added via implementation of new version for the *makeCall* function;

- The Liskov Substitution Principle - no inheritance;

- The Dependency-Inversion Principle - does not have any lower level dependencies;

- The Interface Segregation Principle - composition over inheritance;

- Principles of Least Knowledge - can be invoked only via direct import;

- The Principle of Loose Coupling - exports a single function.

e

10.7 Write Stream

This part of the system implements on function and exports it wrapped into object stream. It receives an object from the down stream with the following structure (Listing: 10.5). It tries to read meta-data of the file with same name as Test Sheet file but with the .js extension, it is possible only if the file exists. In other case, the exception will be thrown and caught by the *catch* block which implements the creation of the file with the content received from the down stream. If the file is already exists then 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 down stream. In case if the modification date of the Test Sheet file is bigger then the modification data of the js file then the system will overwrite the content of the javascript file with the content that was received from the down stream. In case when the Tests Sheet file was not updated after the creation of the representative js file the system will switch to the next object received from the upstream. The result javascript file generated from the provided Test Sheet looks as following (Listing: 10.15)

```
/*
 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);
45  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');
50    E6 = data;
    });
    A7.push.call(this, E5, function(err, data) {
      if (err) return (err, null);
      makeComparisonAndWriteResult(E7, data, '|', 'stack push', 'demo/
          demo.xlsx', 'E7');
55    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;
60      A11.push.call(this, E10, function(err, data) {
          if (err) return (err, null);
          makeComparisonAndWriteResult(E11, data, '||', 'stack push',
              'demo/demo.xlsx', 'E11');
          E11 = data;
        });
65    });
    });
  });
  A8.pop.call(this, function(err, data) {
    if (err) return (err, null);
70  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');
75    E9 = data;
```

```
  });
});
```
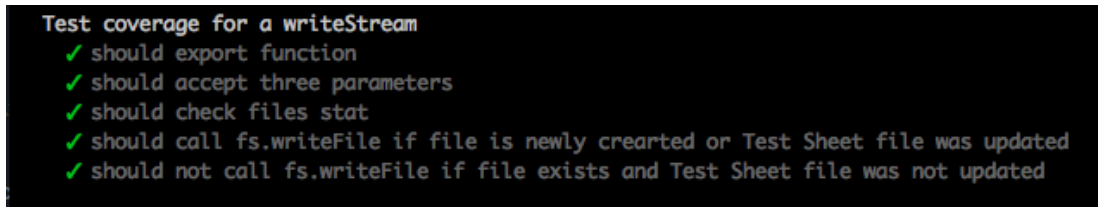
Listing 10.15: Generated javascript file

10.7.1 Test Coverage

The following test cases can show each step incrementally performed during the development process. All calls to systems I/O are implemented using stubs.

The result of tests execution for this module is following (Fig.: 10.8):

Figure 10.8: Test coverage for write_stream.js

10.7.2 Correspondence to the design principles

- Closing - module exports the single stream object with the wrapped function.

- Code to an Interface - calls are done to the public functions of the required objects.

- Do not repeat yourself - small code duplication in callbacks for file writing, while avoiding this will add more complexity;

- The Single-Responsibility Principle - can be changed due to the introduction of requirements for support of multiple sheets within the single xlsx file.

- The Open/Closed Principle - performs writing to the file only;

- The Liskov Substitution Principle - no inheritance;

- The Dependency-Inversion Principle -

  - The 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;

- The Principle of Loose Coupling - communication is done via standard file system and standard stream interfaces.

10.8 Reporting mechanism

The reporting mechanism made as a standalone npm package which must be installed globally. This is necessary for an ability to be called by generated javascript files described in a previous section. Since they can be located in different parts of the system. 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/

10.8.1 Implementation

The system has following dependencies upon external npm packages:

- Development dependencies:

  – "xlsx": "0.8.0", (`https://www.npmjs.com/package/xlsx`)

- Testing dependencies:

  – "mocha": "2.3.4", (`https://www.npmjs.com/package/mocha`)

– "sinon": "1.17.2", (`https://www.npmjs.com/package/sinon`)

```
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});
  var path = file.split('/');
  var resFile = "res_" + path[path.length - 1];
  path[path.length - 1] = resFile;
  var resPath = path.join('/');
  if(!result){
    testSheet.Sheets.Sheet1[variable].v = JSON.stringify(expected) +
        '\\' + JSON.stringify(returned);
    try {
      fs.writeFileSync(resPath, '', { flag: 'wx' });
    } catch (e) {
      if (e.code = 'EEXIST')
        xlsx.writeFile(testSheet, resPath, {'cellStyles': true});
    };
    xlsx.writeFile(testSheet, resPath, {'cellStyles': true});
  };

  return(result);
};
```

Listing 10.16: Report mechanism's main function

On Figure 10.9 you see the result Test Sheet created after execution of the generated demo.js file. Note that on line 10 the *pop* method returns value {*"el": 1* } but not the { *"el": 5* }. This is correct behavior since all the operations are asynchronous and test step 10 does not depend on test step 9. The implementation described in this paper does not guarantee sequential execution of independent test steps. As was already said, this is done for optimization reasons, so that test steps which do not depend upon each other can be started for the execution without waiting for previous step execution being completed.

	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 10.9: Result Test Sheet for stack.js

10.8.2 Test coverage

The following test cases can show each step incrementally performed during the development process. All calls to systems I/O are implemented using stubs. The result of tests execution for this module is following (Fig.: 10.10):
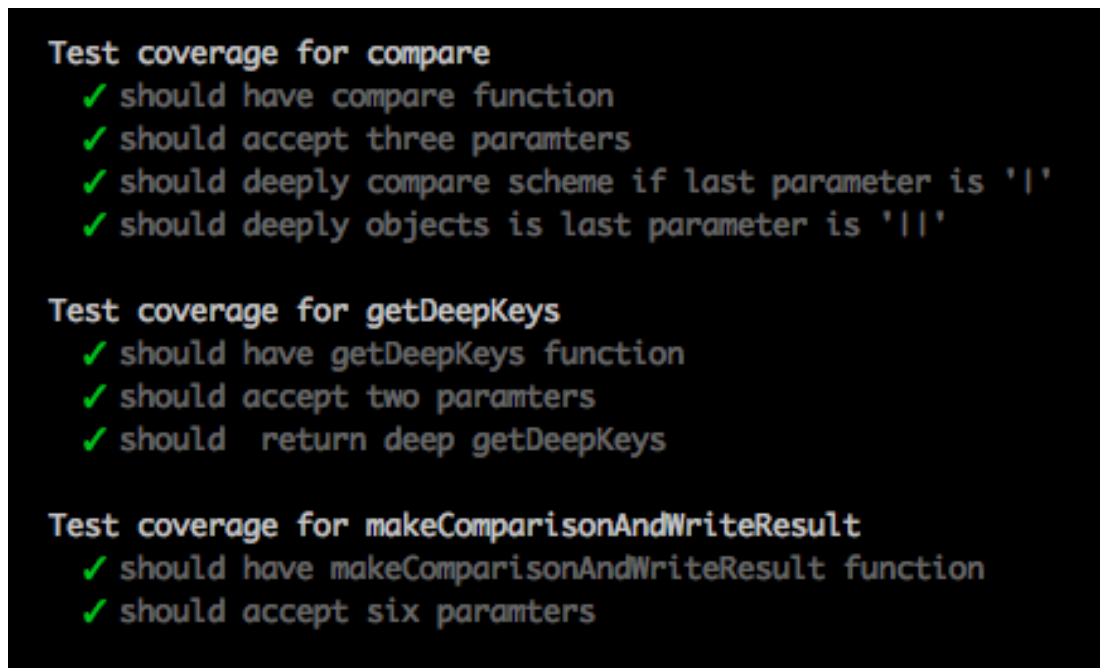
Figure 10.10: Test coverage for report mechanism

10.8.3 Correspondence to the design principles

- Closing - comparison and reporting processes done in the 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 the direct import;

- The Principle of Loose Coupling - exports a single function.

11 Performance

The content of this chapter shows the performance measurements made by the author regarding Test Sheets processing (generation of executable javascript file from the spreadsheet file) and tests execution. All measurements were made on hardware with following characteristics:

- Mac mini (Late 2014)

- **Processor:** 2.6 GHz Intel Core i5;

- **Memory:** 8 GB 1600 MHz DD3;

- **Storage:** 1 TB SATA Disk.

The characteristics of the tests' execution environment are following:

- Operational System:

    - OS X El Capitan

    - version 10.11.4

- node.js environment:

    - node version 4.4.5

    - npm version 2.15.5

11.1 Generation performance

In this section we describe the system's performance characteristics for generating executable javascript files from the Test Sheet xlsx files. The performance was measured for two cases: the first case is when all the Test Sheet files are located in a single (root) directory, the second case when root directory contains Test Sheet files and another folder with Test Sheet files and yet another folder.

Single folder   - all files are located within the same (root) directory (Diagram: 11.1).

- **New** - all the files are newly added on the every iteration of the measurement with deletion of all the previously generated javascript files. Thus the number of generated js files is equivalent to the number of files in a directory.

- **Add** - only ten xlsx files are added on every iteration of the measurement processes. Thus previously added xlsx files are skipped for javascript files creation. All generated files are skipped as files with non xlsx extension.
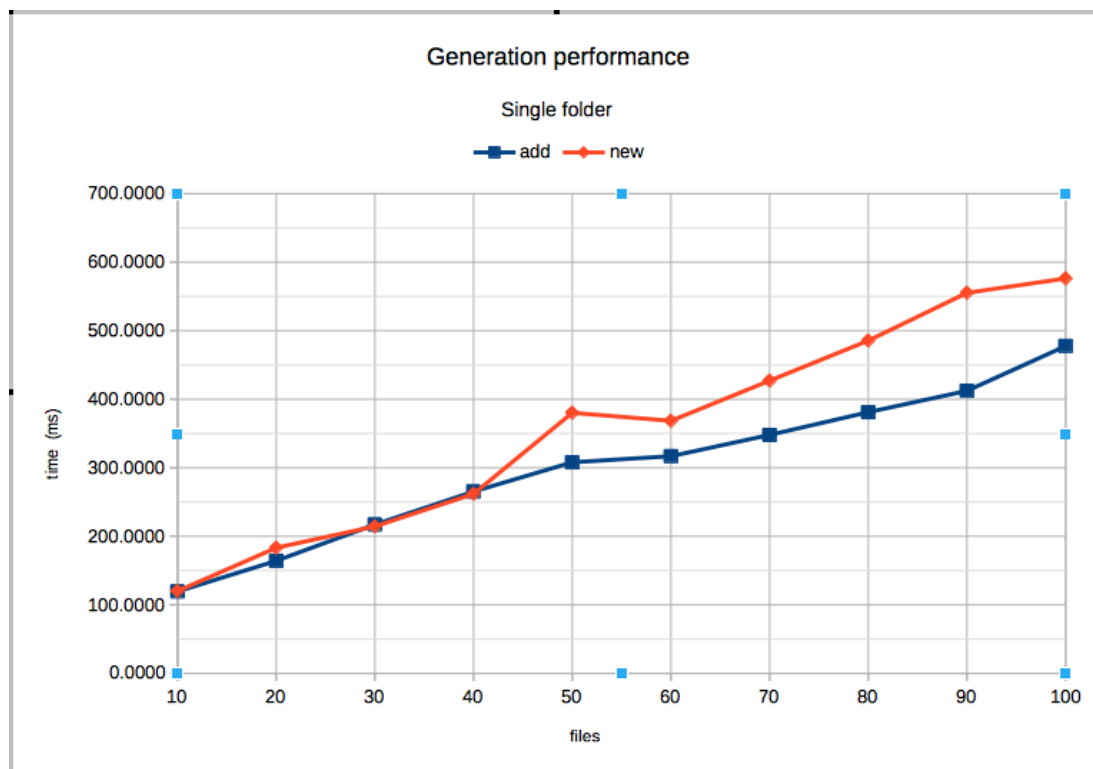
Figure 11.1: Generation performance for files within single folder

Nested folder   - each folder contains ten xlsx files and another folder with the same content (Diagram: 11.1).

- **New** - all the files are newly added on the every iteration of the measurement with deletion of previously generated javascript files. Thus the

number of created javascript files is equivalent to the number of files in a directory and nested folder(s).

- **Add** - ten xlsx files with the folder nested within the folder created inside the directory from the previous iteration are added. Thus previously added xlsx files are skipped for the javascript files creation. The javascript files are generated only for the folder with the deepest level of nesting.
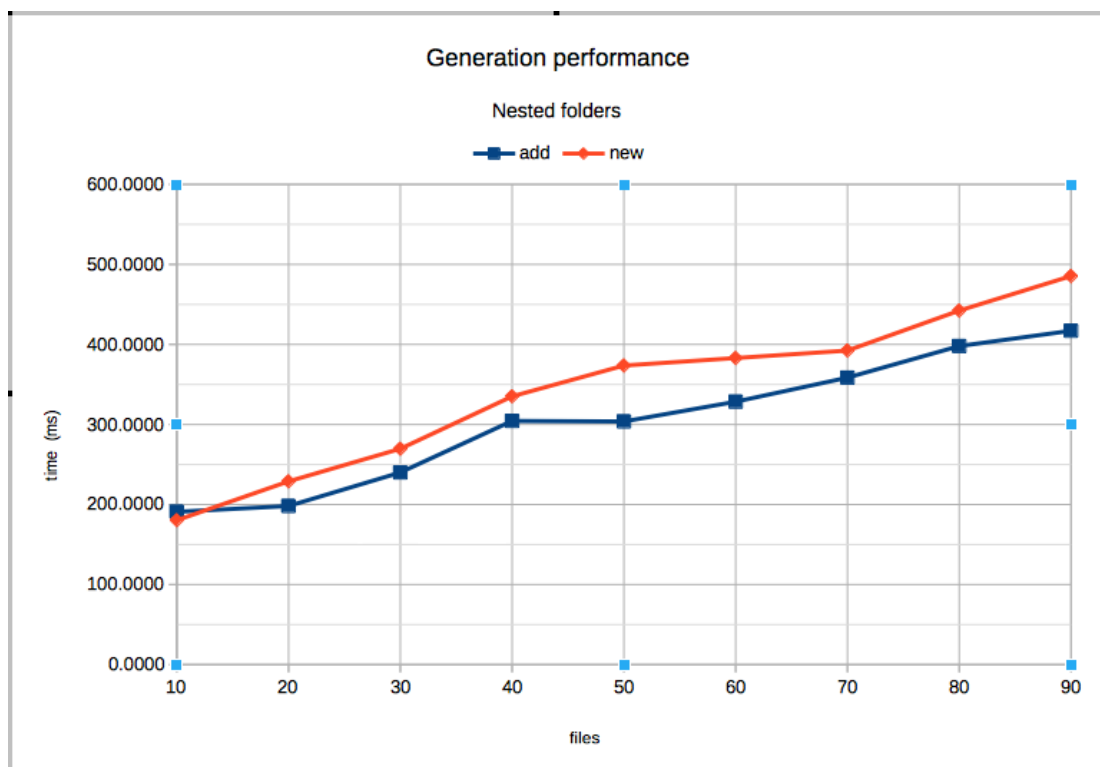
Figure 11.2: Generation performance for files within nested folders

11.2 Executions performance

This section shows execution performance measured for the javascript files generated from the Test Sheet file. All the files are located within the same folder. The measurements were done for two extreme cases of test steps definitions. The first case, when all test step are independent from each other thus can be invoked for execution within the same iteration of the event loop. The second case, when input of all test steps except the first is defined by the result of the previous test

step. Thus, there is only one test step called per event loop iteration and the amount of iterations is equal to the number of test steps. All measurements are done for both deep and scheme only comparisons.

Independent test steps - all test steps are independent from each other. For this reason their calls are done within the same iteration of an event loop (Diagram: 11.2).

- **Scheme comparison** - actual and expected outputs are compared by their scheme

- **Deep comparison** - actual and expected outputs are compared by their scheme and values of every property
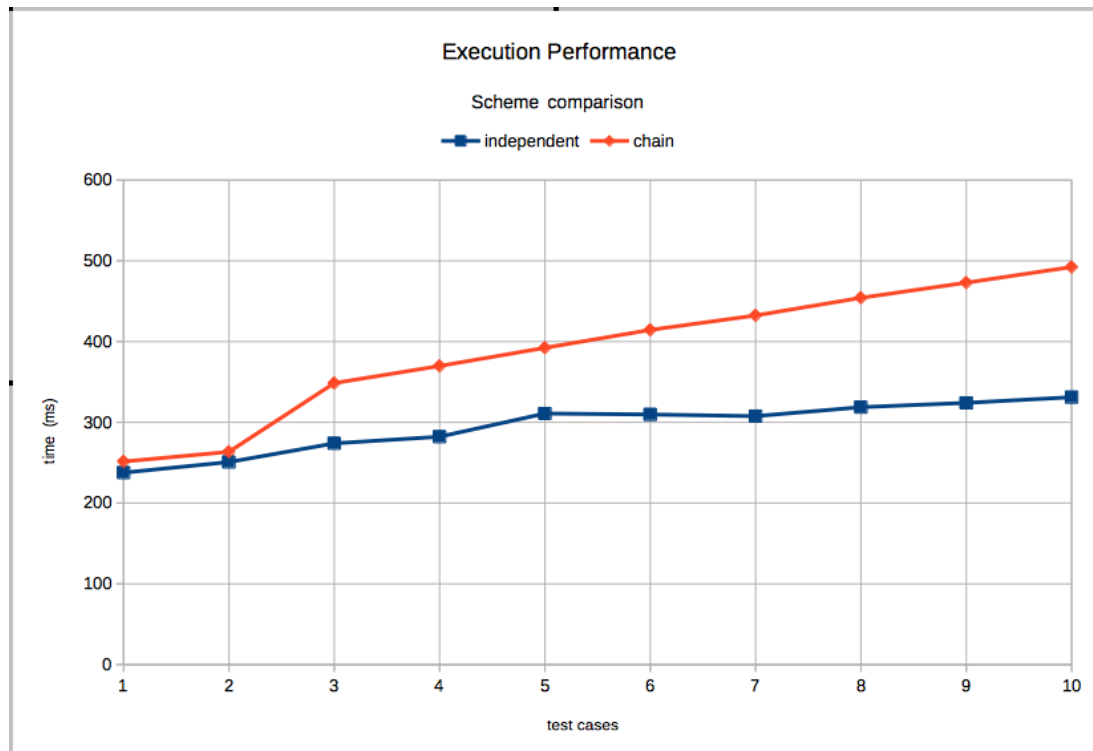
Figure 11.3: Execution performance for independent test steps

Chain relationship - output of each test step (except the first) defines input for row below. Therefore there is one execution call per each iteration of the event loop (Diagram: 11.2).

- **Scheme comparison** - actual and expected outputs are compared by their scheme

- **Deep comparison** - actual and expected outputs are compared by their scheme and values of every property
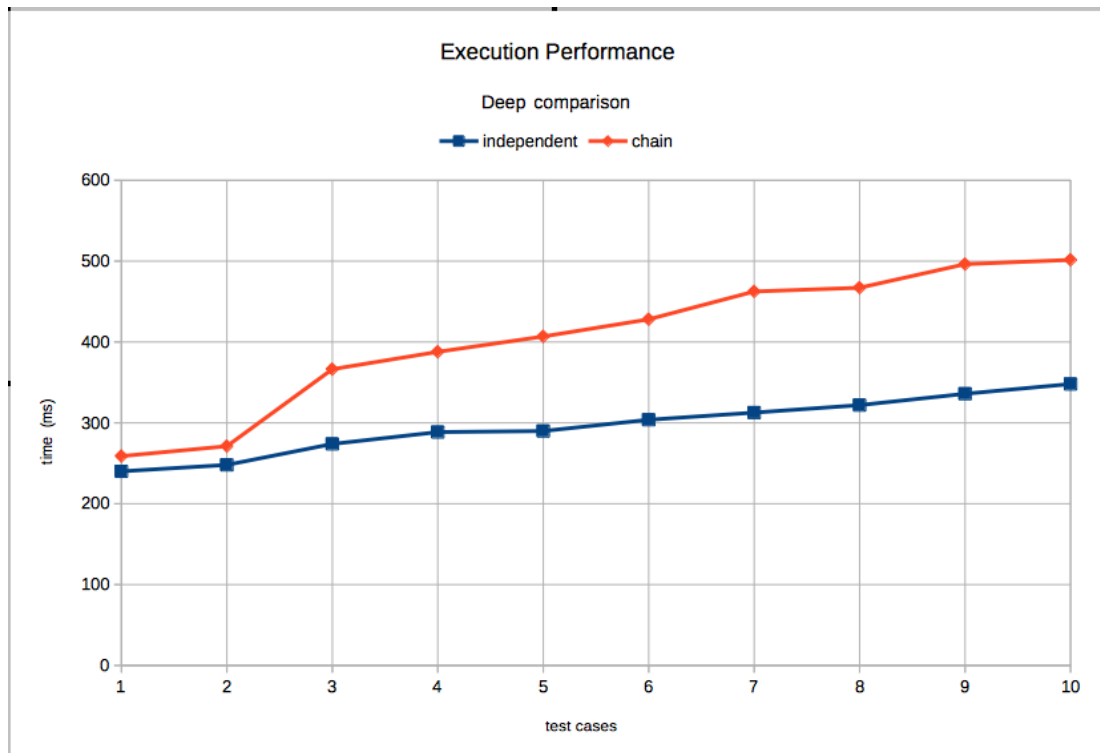
Figure 11.4: Execution performance for interdependent test steps

The measurements from the first section shows that the Test Sheets can be structured within the root folder in arbitrary way without big looses in the generation performance. This should give positive impact on the processes of tests creation and maintenance.

The performance for tests execution proves the necessity of execution order optimization implemented within the *scheme* module. It also shows that type of comparison does not influence the test execution time for small JSON object with simple structure.

12 User Experience

The user experience collection was performed after introducing the final version of the system. First was given a workshop regarding the system use. This decision was taken due to the fact that not all volunteered employees were the target users. Total number of respondents was seven. All of them were from different divisions of the company and had a different background. Three of respondents were from customer support and project management. Another four were from the development team.

The task for respondents during collection of user experience was to recreate test coverage introduced in the Chapter 10 for the module *stack* from the same chapter. They were allowed to use their favorite spreadsheet editor.

The scenario was defined in a following way:

1. Open spreadsheet editor;

2. Create new spreadsheet;

3. In the cell **A1** add the *description* text **User experience**;

4. In the cell **A2** add the *file under test* name **stack**;

5. In the cell **A3** add *module under test* name **stack**;

   - In the cell **B3** add the *method under test* name **size**;

   - In the cell **D3** add the indicator of *deep comparison* ||;

   - In the cell **E3** add the *expected return* value **{ "size": 0 }**;

6. In the cell **A4** add the *module under test* name **stack**;

   - In the cell **B4** add the *method under test* name **push**;

   - In the cell **C4** add the *input parameter* value **{ "el": 1 }**;

   - In the cell **D4** add the indicator of *deep comparison* ||;

   - In the cell **E4** add the *expected return* value **{ }**;

7. In the cell **A5** add the *module under test* name   ***stack***;

   - In the cell **B5** add the *method under test* name ***top***;

   - In the cell **D5** add the indicator of *deep comparison* ||;

   - In the cell **E5** add the <u>reference</u> to the cell **C4** as an *expected return value*;

8. In the cell **A6** add the *module under test* name ***stack***;

   - In the cell **B6** add the *method under test* name ***push***;

   - In the cell **C6** add the reference to th cell **E5** as an *input parameter*;

   - In the cell **D6** add the indicator of *schema comparison* |;

   - In the cell **E6** add the *expected return* value **{ }**;

9. In the cell **A7** add the *module under test* name ***stack***;

   - In the cell **B7** add the *method under test* ***push***;

   - In the cell **C7** add the <u>reference</u> to th cell **E5** as an *input parameter*;

   - In the cell **D7** add the indicator of *schema comparison* |;

   - In the cell **E7** add the *expected return* value **{ }**;

10. In the cell **A8** add the *module under test* name ***stack***;

    - In the cell **B8** add the *method under test* name ***pop***;

    - In the cell **D8** add the indicator of *deep comparison* ||;

    - In the cell **E8** add the *expected return* value **{ "el": 5 }**;

11. In the cell **A9** add the *module under test* name ***stack***;

    - In the cell **B9** add the *method under test* name ***push***;

    - In the cell **C9** add the <u>reference</u> to th cell **E8** as an *input parameter*;

    - In the cell **D9** add the indicator of *deep comparison* ||;

    - In the cell **E9** add the *expected return* value **{ }**;

12. In the cell **A10** add the *module under test* name ***stack;***

    - In the cell **B10** add the *method under test* name   ***pop***;

    - In the cell **D10** add the indicator of *deep comparison* ||;

- In the cell **E10** add the <u>reference</u> to the cell **E5** as an *expected return value*;

13. In the cell **A11** add the *module under test* name **stack**;

    - In the cell **B11** add the *method under test* name **push**;

    - In the cell **C11** add the <u>reference</u> to th cell **E10** as an *input parameter*;

    - In the cell **D11** add the indicator of *deep comparison* ||;

    - In the cell **E11** add the *expected return* value **{ }**;

14. Save the file in to the folder *stack* with name *ux.xlsx*;

15. Generate executable js file running *node test_sheets ./stack* from the terminal;

16. Execute the generated file running *node ./ux.xlsx*;

17. Open *ux.xlsx* file and check test execution results.;

After the task completion every respondent filled up two questioners regarding their user experience.

12.1 After Scenario Questionnaire

The After Scenario Questionnaire (ASQ) devloperd by Lewis [47], was given to a study subject after he/she has completed the scenario. The users circle their answers using the provided seven point scale. The lower the selected score, the higher the subject's usability satisfaction with their system. The ASQ score is calculated by taking the arithmetic average of the all questions.

The grades represented in the table are arithmetical average for each question from all results with three decimals precision.

12.2 Post Study System Usability Questionnaire

The second questionnaire provided to the respondents was Post Study System Usability Questionnaire (PSSUQ) developed by Lewis [47]. Like the ASQ, the PSSUQ requires users circle their response to each question based a seven point

Property	Value
The ease of completing this task.	1.714
The amount of time it took to complete this task	2
The support information	2.571
ASQ	2.095

Table 12.1: After Scenario Questionnaire

scale. The lower the response, the higher the subject's usability satisfaction. The list of questions:

1. *Overall, I am satisfied with how easy it is to use this system.*

2. *It was simple to use this system.*

3. *I could effectively complete the tasks and scenarios using this system.*

4. *I was able to complete the tasks and scenarios quickly using this system.*

5. *I was able to efficiently complete the tasks and scenarios using this system.*

6. *I felt comfortable using this system.*

7. *It was easy to learn to use this system.*

8. *I believe I could become productive quickly using this system.*

9. *The system gave error messages that clearly told me how to fix problems.*

10. *Whenever I made a mistake using the system, I could recover easily and quickly.*

11. *The information (such as on-line help, on-screen messages and other documentation) provided with this system was clear.*

12. *It was easy to find the information I needed.*

13. *The information provided for the system was easy to understand.*

14. *The information was effective in helping me complete the tasks and scenarios.*

15. *The organization of information on the system screens was clear.*

16. *The interface of this system was pleasant.*

17. *I liked using the interface of this system.*

18. *This system has all the functions and capabilities I expect it to have.*

19. *Overall, I am satisfied with this system.*

The PSSUQ is used to produce the following measures:

- **Overall user satisfaction with their system** (OVERALL) – calculated by taking the average of questions 1-19;

- **System usefulness** (SYSUSE) – calculated by taking the average of questions 1-8;

- **Information quality** (INFOQUAL) – calculated by taking the average of questions 9-15;

- **Interface quality** (INTERQUAL) – calculated by taking the average of questions 16-18;

The number taken for measures calculations were the arithmetic mean of answer values for each question.

Measure	value
OVERALL	2.000
SYSUSE	2.000
INFOQUAL	2.061
INTERQUAL	2.190

Table 12.2: Performance comparison of patterns for asynchronous information flow

Values of ASQ, OVERALL and SYSUSE show high level of user satisfaction. First of all, this was achieved by the fact the concept of Test Sheets allows users to apply any spread sheet editor. The next factor is relative simplicity of the module under test and consequently the test coverage users were asked to recreate. Further, the conventions introduced for current implementation are straight forward and simple. Despite of this there is a room for user experience improvement. The most complains regarding the system's use were about necessity of using command line interface instead of graphical user interface.

13 System fit and benefits

13.1 Task-Technology Fit

Task Technology Fit (TTF) research introduced by Goodhue and Thompson [48] combines theories of job satisfaction and performance. It argues that the effect of technology on the performance cannot be measured independently from the task the technology is supposed to support. This complex model takes in account characteristics of the task itself, the technology which helps to accomplish this task and an individual who has to perform this task. (Figure: 13.1).
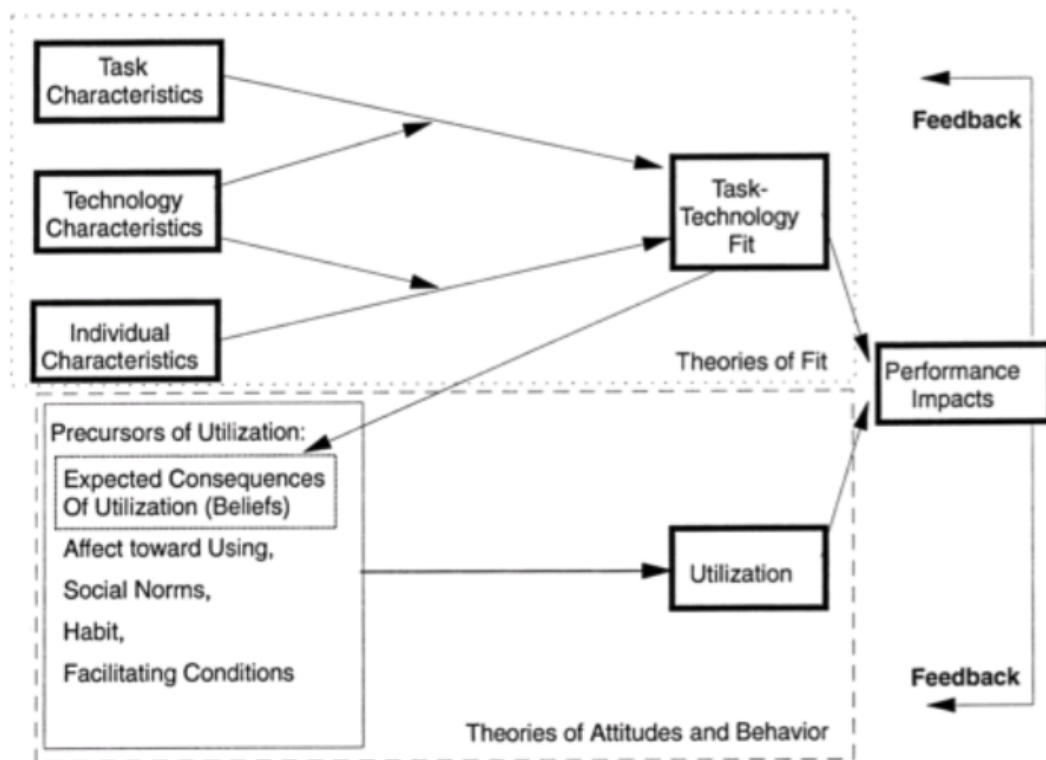
Figure 13.1: Task-Technology Fit model[48]

In case of the pilot stage of the Test Sheet in figo GmbH the properties of TTF

model are following:

Task characteristics:

- Moving responsibility for test definition from developers to management staff;

- Reduce the time for bugs allocation and fixation.

Technology Characteristics:

- Asynchronous event driven framework;

- Real-time software.

Individual Characteristics:

- No or little software development background.

The initial assumption was that Test Sheets can be the most beneficial approach to define test steps for system behavior validation. This was done with following "first principles":

Task-Technology-Fit:

- Test Sheet is a representation for tests 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[28];

- node.js is an asynchronous event-driven framework designed to build scalable network applications;

- Optimization for real-time execution is done by changing test steps execution order and position in the event loop with the respect to their reference structure;

- Automatization of bug reporting process directly to the developer (not covered in this researched).

Utilization:

- User experience inherited from spread sheet editors;

- Simple conventions.

Performance Impact:

- Better / automated quality assurance process;

- Wider spread of tasks and responsibility distributions amongst developers and managers;

- Increased awareness / technical understanding of product functionality for managers;

- Time for training and using the test sheet required from managers;

- Change of job descriptions / new tasks and responsibilities for managers.

13.2 Benefits and affecting factors

The factors influenced project's benefits were analyzed from the perspective of short term model (e. g. Functional fit, Overcoming organizational inertia).

Functional Fit   is the extent to which the functional capabilities are embedded and configured within an system package match the functionality that organization needs to operate effectively and efficiently.

- reduced load of developers;

- reduced time for issues detection.

Overcoming Organizational Inertia   is the extent to which members of the organization have been motivated to learn, use, and accept new system. During initial implementation and subsequent upgrade projects, considerable change-management effort, training, and support are needed to overcome organizational inertia.

- high attendance to the workshop and results of user experience measurements showed wills of individuals for system's use within the everyday business process.

The benefits detected after the pilot stage of the Test Sheet project were classified as *operational, managerial, organizational* by their perspectives of the system's influence.

Operational Benefits:

- Feedback cycle time reduction;

- Productivity improvement - Developers can concentrate more on creating and fixing code instead of running manual tests

- Quality improvement - Errors are identified earlier due to automated monitoring, bugs are easier identified before deployments

- Customer Service improvement - The early change detection allows to perform necessary change to scraping scripts without waiting for complains from customers.

Managerial benefits:

- Resource management - No additional quality assurance staff necessary.

Organizational benefits from the system use from the perspective of senior management, is an overall measure of senior managemt's perception of the benefits from the IT-based application. Such benefits usually revolve around the software enabling the faster and more accurate process of coordination and execution, resulting in more tightly controlled organizational processes improved asset utilization and decision making [49]. Next, organizational benefit were observed by chief technical and chief product managers during the use stage of the Test Sheet project:

- Changing work patterns - Reduced workload on developers, increased product quality responsibility for managers

- Facilitating organizational learning - project mangers and customer support staff gain more technical understanding and feel empowered by owning the responsibility for the testing

13.3 Misfits

This section states the cases from the misfits' perspective of the software product implemented in scope of this research and the Test Sheet project. There are six perspectives where misfits can occur: *functionality, data, usability, role, control* and *organizational culture* [48].

Functionality misfits: occur, when the way processes are executed using the software leads to reduced efficiency and effectiveness compared to pre-ES outcomes. For this project no functionality misfits were detected.

Data misfits: occur, when data or its characteristics stored in or needed by the software leads to data quality issues such as inaccuracy, inconsistent representation, inaccessibility, lack of timeliness or inappropriateness for users' context. None of such cases was detected during system use.

Usability misfits: occur, when the interactions with the software required for task execution are cumbersome or confusing. During the system users' complains were generalized to the following statements:

- It is necessary to manually set define JSON object to be sent to API as well as object for an expected result;

- Test generated code must be copied to the same machine as working system (only for testing of scraping scripts)

- Code generation requires call execution of the command from the terminal.

Role misfits occur, when the roles in the software are inconsistent with the skills available, create imbalances in the workload leading to bottlenecks and idle time, or generate mismatches between responsibility and authority. During the system's use none of such cases was detected.

Control misfits   occur, when the controls embedded in the software provide too much control, inhibiting productivity, or too little control, leading to the inability to assess or monitor performance appropriately. The only misfit detected from this perspective was identified on a project planning stage and was accepted as a necessary trade off by the technical management team:

- Users need to have an access to the production version of the code base to define test cases;

Organizational culture misfits   occur, when the software requires ways of operating that counterforce organizational norms. Within the system use period there was no users' or any other staff complains regrading any of this problems.

The most of misfits detected during the system use are related to the usability factor. Input and output parameters JSON type required by the nature of system under test. This factor can by overcome by defining more convenient structure from the users' perspective and adding internal translation from a new data type to JSON. Generation of executable files as well as their deploying to the production environment planned to be done with background task which uses version control system's hooks.

Despite of the factors stated above the user experience is good. As well as the overall TTF. This was achieved by the Test Sheet approach itself, event system of node.js. Furthermore, the approach taken for code generation which reduce idle time of the generated code and reporting mechanism (additional feature which lays outside of the research scope). The benefits of the system use have showed themselves already during the pilot stage in all operational, managerial and organizational layers.

14 Limitations and Future Work

This paper provide the prove of concept for usage of Basic Test Sheets only for asynchronous testing of real-time software. The same process can be applied for Higher-Order/Parametrized Test Sheets in a straight forward way since such test sheets just provide additional level of data abstraction without changing generation and execution processes.

However, for the Non-linear Test Sheets this task is not so trivial. The reason for it is the order of test execution defined by the finite automates. Which necessitates changes into the algorithm for execution order designed by the author. In this case, two uncertainties are laying on the surface. First, which event should be taken into account for the calculation of counts the one test step was executed (call event or return event). Second, how to minimize system's time in idle state while it is waiting for response from the external system.

Further, described implementation requires system under test to have internal logic for handling the execution time. As a future work it is possible to add column for putting additional constraints for execution time of a function under test.

Due to the optimization and nature of asynchronous software some test steps can be started without previous steps being completed which can lead to confusing test assumptions. This can be avoided by making test steps dependent upon previous steps either directly or via additional intermediate test steps.

The implementation of system described in this paper is bounded to node.js framework, which does not allow to use the main benefit of javascript language, an ability to be directly executed inside of the web browser. In such case the browser plug-in for Test Sheets can be developed to work with cloud-based spreadsheet editors. Three changes should be done to make it possible. The first change will lay in the connector library for reading spreadsheet files, same should be done in case of changing the spreadsheet editor. Next, the translation software must

be compiled to reduce the load on the browser's javascript engine. Furthermore, the compilation has to be done over generated javascript files and they must be recorded into the browser's cache.

Further, all the code covered by tests should be located on the computer where generated javascript files are stored with relative paths to the modules under test. This caused by the fact that node.js' require module works only with relative paths (with the exception for globally installed packages). Moreover system uses standard fs module for I/O operations, but it is the common case to store the code within remote repositories. In combination with the web-browser plug-in it is possible to create the system for remote work with code base based on any device which has an Internet connection and able to run web browser which supports javascript.

Moreover, as was shown by the user experience, the command line interface is confusing for people without software development background. Therefore, either executable file of graphical user interface with an ability to call Test Sheets' files processing shall be created.

Last but not least, all asynchronous functions covered by tests implements callback for handling events. However it is possible that the function with asynchronous behavior returns promise object for deferred value, in such case the implementation of template library and reporting mechanism should be changed.

15 Conclusions

The software product developed during the Test Sheet project in figo GmbH is a prove of concept for use of the Test Sheet approach for asynchronous testing of real-time systems.

The callback pattern for handling asynchronous events together with code generation optimization which allows to call independent test steps within the same iteration of an event loop makes it possible to run the tests on a timely basis.

The design and architecture of the system makes it easy to enhance and change. While application of test driven development approach provides control over the code.

Conventions developed for this project expand the original Test Sheets approach with an opportunity to perform two type of comparison for test steps results.

User experience collected during the pilot stage of the project shows that Test Sheets is comfortable and convenient approach to define test for people with little or without software development background.

Analysis of misfits and benefits of the system's use indicates the practical applicability of the Test Sheets approach for asynchronous testing of the real-time software systems.

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] A case study of toyota unintended acceleration and software safety. `https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf`.

[7] Casperjs. `http://casperjs.org/`.

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

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

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

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

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

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

[14] Fit. `http://fit.c2.com/`.

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

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

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

[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] Javascript — mdn. `https://developer.mozilla.org/en-US/docs/Web/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] Phantomjs - scriptable headless webkit. `https://github.com/ariya/phantomjs/`.

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

[27] Real-time system references and reading materials. `http://www.ece.ubc.ca/~gillies/readings.htm`.

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

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

[30] The unix oral history project. `http://www.princeton.edu/~hos/Mahoney/expotape.htm`.

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

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

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

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

[35] 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-`.

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

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

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

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

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

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

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

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

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

[45] Robert L. Glass. Real-time: The “lost world” of software debugging and testing. *Commun. ACM*, 23(5):264–271, May 1980.

[46] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[47] James R. Lewis. Ibm computer usability satisfaction questionnaires: Psychometric evaluation and instructions for use. *Int. J. Hum.-Comput. Interact.*, 7(1):57–78, January 1995.

[48] Alexander Maedche. L10-intro es use. 2014.

[49] Alexander Maedche. L11-intro es use process value. 2014.

[50] Alexander Maedche. L5-intro es transformation. 2014.

[51] Alexander Maedche. L6-intro es development. 2014.

[52] Alexander Maedche. L7-intro es adoption. 2014.

[53] Alexander Maedche. L9-intro es converion. 2014.

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

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

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

[57] 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, July 28, 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, July 28, 2016   Unterschrift