UNIVERSITY OF MANNHEIM

Using Test Sheets for asynchronous testing of Real Time Software

Master Thesis

submitted: July 2016

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

Student ID Number: 1440397

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

Abstract

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

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

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

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

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

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

Contents

List of Figures

List of Tables

1. Introduction

- What precisely did I answer

  what question did I answer

  why should the reader care

  what larger question does this address

- What is my result

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

  What previous work do I build on

  What precisely and in detail my new result

- Why should the reader believe in my result

  What standard was used to evaluate the claim

  What concrete evidence shows that me result satisfies my claim

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

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

study

2. figo GmbH

2.1. General Information

The figo GmbH's mission is to "build the backbone of next generation financial services"[10], company was founded in 2012 and has its headquarter in Hamburg. Total Equity Funding: $778.660 in 3 Rounds from 7 Investors (November 3, 2015). Currently, the API is fully functional in Germany and partly in Austria.[6][7]

figo Connect API was created to accelerate innovation in the FinTech area and to allow figo's partners to offer products with real added value. It enables developers, startups and even banks to connect to every financial service provider. These partners can access every bank account (current, savings, loan, securities, ...), credit card, eWallet and other financial services like PayPal through one single REST-API. [24][10][27]

Functions available through API:[8]

- Create, Read, Update, Delete Bank account(s);
- Read, Update, Delete Bank account(s) transactions;
- Create, Read, Update, Delete Bank account(s) standing orders;
- Create, Read, Update, Delete Bank account(s) direct debits;
- Read, Update Bank account(s) securities;
- Create, Read, Update, Delete Bank payment(s);

List of partners of figo API with their use-cases: `http://figo.io/use_cases.html`

2.2. IT infrastructure

Information technology infrastructure of figo GmbH has two parts related to the external connections. **API Server** - implements interfaces to figo's customers

and partners for accessing banking inforamtion and services. **Banking Server** - implements connection to banks via three possible communication channels. This particular part lays in scope of this research as a part where implemented system is running. Below provided description introduces basic concepts of Banking Service Architecture. For more information regarding IT infrastructure please use (pics with arch scheme attached)

2.2.1. Banking Server Architecture

Custom-API. Introduction of Directive on Payment Services (PSD) and PSD2 by European Commission in European Union and initiatives of Government in United Kingdom regarding provision of API by Banks and its standartization obligated with providing of online access points to their services.[15][17][3] Some of the banks already implemented Application Program Interfaces to access their services. Some of which provide full functionality while some only partial. Within the Single European Payment Area acceptance of directive by European Bank Authority scheduled within 2017 year. [5] All this APIs very in their structure and functionality, and connection process lays out of the scope of this research.

Moreover some of the banks implements standardized programmable entry point.

HBCI+/FinTS - bank-independent protocol for online banking, developed and used by German banks. Home Banking Computer Interface (HBCI) was originally designed by the two German banking groups Sparkasse and Volksbanken und Raiffeisenbanken and German higher-level associations as the Bundesverband deutscher Banken e.V.. The result of this effort was an open protocol specification, which is publicly available. The standardisation effort was necessary to replace the huge number of deprecated homemade software clients and servers (some of them still using BTX emulation). While IFX (Interactive Financial Exchange), OFX (Open Financial Exchange) and SET are tailored for the North American market, HBCI is designed to meet the requirements of the European market.[9]
Features[9]:

- Support for online-banking using PIN/TAN one time passwords.

- Support for online-banking with SWIFT.
- DES and RSA encryption and signatures.
- Making use of XML and SOAP for data-exchange, encryption and signatures.
- Implemented on top of HTTP, HTTPS and SMTP as communication layer.
- Multibanking: The software clients are designed to support accounts on multiple banking companies.
- Platform Independency: The specification allows software development for various types of clients.
- Storage of the encryption keys on an external physical device (smart card) for improved security.
- Possibility to use so called "Secoder" smart card readers to allow the user to cross check the transaction data on a secure device before signing it to uncover manipulations caused by malware. To use Secoder the bank as well as the home banking software have to support the Secoder protocol extension of FinTS

Low level vocabulary for message communication is defined by ISO20022 for more information please have a look to `https://www.iso20022.org/`

Web-Banking Engine. Since not all of the banks provide API nor HBCI figo uses web scraping technology (implemented within Web-Banking Engine) to access account's information and perform interaction with internet banking web page. From the banks perspective interaction completely looks like directly with user, while user does not feel the difference between interaction via Custom-API or HBCI or Web-Banking Engine.
In a same time this is the most sensitive part from the developer's perspective since every change to the bank's web page can leads to failure of the specific scripts. The critical part to recognize such significant for scraping changes before any user's interaction and notify a developer regarding part of the script which failed. Exactly for this purpose testing scripts generated from Test Sheets are used as a (demon task/crown job) in real time fashion with recording unexpected behavior to the general log and notification of the developer (email/slack).

Next chapter provides general information regarding web scraping and high-level description of scraping tool used by figo Gmbh.

3. Testing

" Test processes determine whether the development products of a given activity conform to the requirements of that activity and whether the system and/or software satisfies its intended use and user needs. Testing process tasks are specified for different integrity levels. These process tasks determine the appropriate breadth and depth of test documentation. The documentation elements for each type of test documentation can then be selected. The scope of testing encompasses software-based systems, computer software, hardware, and their interfaces. This standard applies to software-based systems being developed, maintained, or reused (legacy, commercial off-the-shelf, Non-Developmental Items). The term "software" also includes firmware, microcode, and documentation. Test processes can include inspection, analysis, demonstration, verification, and validation of software and software-based system products."[11] "Software testing has grown as an important technique to evaluate and help to improve software quality. Numerous techniques and tools have appeared in the last decade, ranging from static analysis to automatic test generation and application. One can argue that software is the dominant part of an embedded system, either as a final product (executable code) or during its development lifecycle (system modeling in specific languages and computation models). In both cases, software must be thoroughly verified to ensure product quality and reliability. One can observe a growing number of academic and industrial works on the topic of embedded SW testing in the last four or five years, and this seems to be a good time for reflection: how exactly is embedded software testing different from traditional software testing? Is it an engineering or computer science problem? Does it need extra support from platform developers? What is the role of the SW engineer and of the designer in developing a high-quality software-based embedded application? Many authors suggest that, on top of the ordinary software testing challenges, software usage in an embedded application brings additional issues that must be dealt with: the variety of possible target platforms, the different computational models involved during the design, faster time-to-market and even more insta-

ble and complex specifications, platform-dependent constraints (power, memory, and resources availability), etc. On the other hand, current platforms are more and more powerful, and the specificities of the embedded application can help to reduce the search space during test generation: application domains, strong code reuse paradigm, use of less advanced programming language resources, and common availability of system models subject to or already verified with respect to specific properties, for instance. Furthermore, a major part of the so-called embedded software does not depend directly on hardware, and one can argue that only a small percentage really needs to be tested together with the target platform, and this test is part of the platform design, not the system design. "[34]

Introduction of Agile and Extreme development paradigms with increased requirements for control over the existing code, its reuse and maintenance lifted requirements for tests and their quality to the new level and shifted responsibility regarding testing process from software engineers to people without development background. This led to new requirements for testing definition and representation tools since old tools like JUnit have high entry level for test engineers and requires its users to have an ability to write and read an executable code in a corresponding programming language.

"Tests are as important to the health of a project as the production code is. Perhaps they are even more important, because tests preserve and enhance the flexibility, maintainability, and reusability of the production code. "[41]

"The ideas and techniques of software testing have become essential knowledge for all software delopers"[29]

Testing is responsible for at least 30% of the cost in a software project.[32]

NIST 2002 report, "The Economic Impacts of Inadequate Infrastructure for Software Testing" claimed that - inadequate software testing costs the US alone between $22 and $59 billion annually. [31]

Huge losses due to web application failures Financial services : $6.5 million per hour (just in USA!) Credit card sales applications : $2.4 million per hour (in USA) [31]

3.1. Test Driven Development

Test Driven Development is a software development paradigm which combines test-first development and refracting. This means that programmers first define test and only after write code to fulfill test requirements, both code and test maintained by developers. This reduces amount of redundant code and improves developers' confidence during refractoring process. Refactroing in TDD requires developer before introducing new feature first ask question the existing design fits best for implementation of new functionality.[12] [30]

"Consider the following three laws: 1)First Law You may not write production code until you have written a failing unit test. 2)Second Law You may not write more of a unit test than is sufficient to fail, and not compiling is failing. 3)Third Law You may not write more production code than is sufficient to pass the currently failing test.

These three laws lock you into a cycle that is perhaps thirty seconds long. The tests and the production code are written together, with the tests just a few seconds ahead of the production code. If we work this way, we will write dozens of tests every day, hundreds of tests every month, and thousands of tests every year. If we work this way, those tests will cover virtually all of our production code. The sheer bulk of those tests, which can rival the size of the production code itself, can present a daunting management problem.[41]

Pros. As a result you will always be improving the quality of your design, thereby making it easier to work with in the future.[12] Excelent fault isolation. Support by big variety of test frameworks [39] Tests can replace code documentation at certain level of abstraction [41]

Cons. Very hard to perform TDD without testing framework. [39] Also [39] states among the limitations of TDD is its bottom-up nature which provides little oportunity for elegant design. In contrast [12] citates Robert C Martin with following words: "The act of writing a unit test is more an act of design than of verification. It is also more an act of documentation than of verification. The act of writing a unit test closes a remarkable number of feedback loops, the least of which is the one pertaining to verification of function" For the same bottom-up

approach [39] claims that some of the faults can be tracked only by data flow testing.

3.2. Requirements for tests (F. I. R. S. T.)

Fast. Tests should be fast. They should run quickly. When tests run slow, you won't want to run them frequently. If you don't run them frequently, you won't find problems early enough to fix them easily. You won't feel as free to clean up the code. Eventually the code will begin to rot.[41]

Independent. Tests should not depend on each other. One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like. When tests depend on each other, then the first one to fail causes a cascade of downstream failures, making diagnosis difficult and hiding downstream defects.[41]

Repeatable. Tests should be repeatable in any environment. You should be able to run the tests in the production environment, in the QA environment, and on your laptop while riding home on the train without a network. If your tests aren't repeatable in any environment, then you'll always have an excuse for why they fail. You'll also find yourself unable to run the tests when the environment isn't available.[41]

Self-Validating. The tests should have a boolean output. Either they pass or fail. You should not have to read through a log file to tell whether the tests pass. You should not have to manually compare two different text files to see whether the tests pass. If the tests aren't self-validating, then failure can become subjective and running the tests can require a long manual evaluation.[41]

Timely. The tests need to be written in a timely fashion. Unit tests should be written just before the production code that makes them pass. If you write tests after the production code, then you may find the production code to be hard to test. You may decide that some production code is too hard to test. You may not design the production code to be testable.[41]

(MORE SHOULD BE ADDED)

3.3. Reliability testing

"Reliability is the probability that a system, or a system component, will deliver its intended functionality and quality for a specified period of "time", and under specified conditions, given that the system was functioning properly at the start of this "time" period. For example, this may be the probability that a real-time system will give specified functional and timing performance for the duration of a ten hour mission when used in the way and for the purpose intended. Since, software reliability will depend on how software is used, software usage information is an important part of reliability evaluation. This includes information on the environment in which software is used, as well as the information on the actual frequency of usage of different functions (or operations, or features) that the system offers. The usage information is quantified through operational profiles."[44]

4. Test Sheets

"Software testing is an important aspect of modern software development. Testing is performed to ensure that a product or component meets the requirements of all stakeholders. Just as the requirements themselves tests can vary widely in their nature. Some tests check whether executing certain code paths results in a correct state or answer while others may check whether a certain code path delivers its results in a specified amount of time.

However writing and evaluating tests is often not much different from programming itself. Tests are usually written in a formal programming language such as Java/JUnit. This necessitates that in order to create a test and understand its results knowledge of a formal programming language is required. This is a significant barrier of entry for stakeholders without a background in IT even though they may be interested in the tests themselves. Even without deep IT knowledge a stakeholder may still be interested in how well a product performs with regard to her requirements.

Visual test representations such as the UML Testing Profile try to lower the barrier of entry into testing. However most of the time these visualizations are only partial descriptions of the tests and so do not contain all the desired information for evaluation.

The Software Engineering group has started to develop a new representation for tests called Test Sheets. The goal is to create a way to define tests which combines 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. This is achieved by representing a test in tabular form as a spreadsheet. Rows in a Test Sheet represent operations being executed while the columns represent input parameters and expected results. The actual content of a cell can be made dependent on other cells by addressing them via their location. This works in a way similar to existing spreadsheet software such as Microsoft Excel. After executing a test the cells for each expected result is colored according

to the result of the test. A successful test causes cells to become green while failed tests are indicated by red cells."[19]

4.1. Basic Test Sheets

"A Test Sheet consists of a name and a class being tested. Each row after that represents one method call. The first column identifies the object being tested while the second column indicates which method is being called on said object. [...] Input parameters are specified in the columns following the method name up to the invocation line. Right after the invocation line the expected return values can be specified."[20]

4.2. High Order, Parameterized Test Sheets

"The actual value used for Parameterized Test Sheets is specified by a Higher-Order Test Sheet as in the example below. The Higher-Order Test Sheet references the Parameterized Test Sheet as the 'class' being tested. On said pseudo-class it invokes the pseudo-method test followed the by the value to be used as parameter. ?C in the Parameterized Test Sheet is replaced by the value defined the third column (column C) for each execution. It is also possible to use more than one parameter. These are defined in the Higher-Order Test Sheet in subsequent columns (D, E, F, etc.) and referenced in the Parameterized Test Sheet via ?D, ?E, ?F, etc"[21]

5. Real Time Software

5.1. Dependability [40]

Dependability is defined as the trustwothiness of a computer system such that reliance can justifiably be place on the service it delivers. The service delivered by a system is its behaviour as it is perceptible by its users(s); a user is another system (human or physical) iwth the former.
Depening on applications inteded for the system, a different emphasis mau be put on the various facets fo dependability, that is dependablility can be veiwed according to different, but complementary properties which enable the attributes of dependability to be defined:

- *The readiness for usage* leads to **availability**
- *The continuity of service* leads to **reliability**
- *The nonoccurence of catastrophic consequences on the environment* leads to **safety**
- *The nonoccirence of unauthorized disclosure of information* leads to **confidentiality**
- *The nonoccurrence of improper information* leads to **integrity**
- *The ability to undergo repeairs and evelutions* leads to **maintainability**

5.2. Real Time Testing

"Real-time software is a software thaht dries a computer which intercats with functioning external devices or objects. It is called real-time because the software actions control activities that are accuring in an ongoing process."[37]

6. Technology

6.1. Web scraping

"Web scraping (web harvesting or web data extraction) is a computer software technique of extracting information from websites. Usually, such software programs simulate human exploration of the World Wide Web by either implementing low-level Hypertext Transfer Protocol (HTTP), or embedding a fully-fledged web browser, such as Mozilla Firefox.

Web scraping is the process of automatically collecting information from the World Wide Web. It is a field with active developments sharing a common goal with the semantic web vision, an ambitious initiative that still requires breakthroughs in text processing, semantic understanding, artificial intelligence and human-computer interactions. Current web scraping solutions range from the ad-hoc, requiring human effort, to fully automated systems that are able to convert entire web sites into structured information, with limitations." [25]

"The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents." [23]

6.1.1. CasperJS

[4] CasperJS is an open source navigation scripting & testing utility written in JavaScript for the PhantomJS WebKit headless browser and SlimerJS (Gecko). It eases the process of defining a full navigation scenario and provides useful high-level functions, methods & syntactic sugar for doing common tasks such as:

- defining & ordering browsing navigation steps
- filling & submitting forms

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

6.2. NodeJS and Ansynchronous programming

6.2.1. NodeJS

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

"Node.js is considered by many as a game-changer—the biggest shift of the decade in web development.[...]
First, Node.js applications are written in JavaScript, the language of the web, the only programming language supported natively by a majority of web browsers. [...]
The second revolutionizing factor is its single-threaded, asynchronous architecture. Besides obvious advantages from a performance and scalability point of view, this characteristic changed the way developers approach concurrency and parallelism. [...]
The last and most important aspect of Node.js lies in its ecosystem: the npm package manager, its constantly growing database of modules, its enthusiastic and helpful community, and most importantly, its very own culture based on simplicity, pragmatism, and extreme modularity. "[33]

"JavaScript (JS) is a lightweight, interpreted, programming language with first-class functions. Most well-known as the scripting language for Web pages, many

non-browser environments use it such as node.js and Apache CouchDB. JS is a prototype-based, multi-paradigm, dynamic scripting language, supporting object-oriented, imperative, and functional programming styles."[14]

Non blocking I/O A set of bindings responsible for wrapping and exposing libuv and other low-level functionality to JavaScript.[33]

Non blocking I/O in NodeJS is provided by libuv[1][33]. Which is "libuv is a multi-platform support library with a focus on asynchronous I/O. It was primarily developed for use by Node.js, but it's also used by Luvit, Julia, pyuv, and others."[26]
libuv properties[13]:

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

Node core A core JavaScript library (called node-core) that implements the high-level Node.js API.

V8/Chakra the JavaScript engine originally developed by Google for the Chrome browser/ Microsoft for IE 9 browser"[33]

6.2.2. Event handling

+pick from the book NodeJS asynchronous nature provided by event handler which is an implementation of reactor pattern. Here is the description of process lifecycle[33]:

1. The application generates a new I/O operation by submitting a request to the Event Demultiplexer. The application also specifies a handler, which will be invoked when the operation completes. Submitting a new request to the Event Demultiplexer is a non-blocking call and it immediately returns the control back to the application.
2. When a set of I/O operations completes, the Event Demultiplexer pushes the new events into the Event Queue.

3. At this point, the Event Loop iterates over the items of the Event Queue.
4. For each event, the associated handler is invoked.
5. The handler, which is part of the application code, will give back the control to the Event Loop when its execution completes. However, new asynchronous operations might be requested during the execution of the handler, causing new operations to be inserted in the Event Demultiplexer, before the control is given back to the Event Loop.
6. When all the items in the Event Queue are processed, the loop will block again on the Event Demultiplexer which will then trigger another cycle.

6.3. General Theory of Reactivity

In the context of a computer program, reactivity is the process of receiving external stimuli and propagating events. This is a rather broad definition that covers a wide variety of topics. The term is usually reserved for systems that respond in turns to sensors, schedules, and above all, problems that exist between the chair and keyboard.

The field of reactivity is carved into plots ranging from "reactive programming" to the subtly distinct "functional reactive programming", with acrage set aside for "self adjusting computation" and with neighbors like "bindings" and "operational transforms". Adherents favor everything from "continuation passing style" to "promises", or the related concepts of "deferreds" and "futures". Other problems lend themselves to "observables", "signals", or "behaviors", and everyone agrees that "streams" are a good idea, but "publishers" and "subscribers" are distinct.

In 1905, Einstein created a theory of special relativity that unified the concepts of space and time, and went on to incorporate gravity, to bring the three fundamentals of physical law into a single model. To a similar end, various minds in the field of reactivity have been converging on a model that unifies at least promises and observables.

	Singular	Plural
Spatial	Value	Iterable¡Value¿
Temporal	Promise¡Value¿	Observable¡Value¿

However, this description fails to capture all of the varigated concepts of reactivity. Rather, Rx conflates all reactive primitives into a single Observable type

that can perform any role. Just as an array is an exemplar of an entire taxonomy of collections, promises, streams, and observables are merely representatives of their class of reactive primitives. As the common paraphrase of Einstein goes, everything should be made as simple as possible, but no simpler.

6.3.1. Concepts

For the purpose of discussion, we must establish a vocabulary. Some of these names have a long tradition, or at least some precedent in JavaScript. Some are disputed, borrowed, or fabricated.

A value is singular and spatial. It can be accessed or modified. If we break this atom, it will fall into two parts: the getter and the setter. Data flows in one direction, from the setter to the getter.

The duality of a getter and a setter, a producer and a consumer, or a writer and a reader, exists in every reactive primitive. Erik Meijer shows us the parallelism and reflection that exists between various types of reactive duals in his keynote for Lang.NEXT, 2014.

Singular is as opposed to plural or multiple. An array, or generally any collection, contains multiple values. An iterator is a plural getter. A generator and iterator form the plural dual for values in space.

Spatial is as opposed to temporal. Reactivity is about time.

A promise is a getter for a single value from the past or the future. In JavaScript, and in the language E from which we borrowed the concept, the corresponding setter is a resolver. Collectively, an asynchronous value is a deferred.

If a promise is the temporal analogue of a value, a stream is the temporal analogue of an array. The producer side of a stream is a writer and the consumer side is a reader. A reader is an asynchronous iterator and a writer is an asynchronous generator.

Interface			
Value	Value	Singular	Spatial
Getter	Getter	Singular	Spatial
Setter	Setter	Singular	Spatial
Array	Value	Plural	Spatial
Iterator	Getter	Plural	Spatial
Generator	Setter	Plural	Spatial
Deferred	Value	Singular	Temporal
Promise	Getter	Singular	Temporal
Resolver	Setter	Singular	Temporal
Stream	Value	Plural	Temporal
Reader	Getter	Plural	Temporal
Writer	Setter	Plural	Temporal

6.3.2. Singular and plural

An observer can subscribe to eventually see the value of a promise. They can do this before or after the promise has a value. Any number of observers can subscribe multiple times and any single observer can subscribe to the same promise multiple times.

As such, promises model dependency. Promises and resolvers can be safely distributed to any number of producers and consumers. If multiple producers race to resolve a promise, the experience of each producer is indistinguishable regardless of whether they won or lost the race. Likewise, if multiple consumers subscribe to a promise, the experience of each consumer is indistinguishable. One consumer cannot prevent another consumer from making progress. Information flows in one direction. Promises make reactive programs more robust and composable.

Promises are broadcast.

The law that no consumer can interfere with another consumer makes it impossible for promises to abort work in progress. A promise represents a result, not the work leading to that result.

A task has mostly the same form and features as a promise, but is unicast by default and can be cancelled. A task can have only one subscriber, but can be

explicitly forked to create a new task that depends on the same result. Each subscriber can unsubscribe, and if all subscribers have unsubscribed and no further subscribers can be introduced, a task can abort its work.

Tasks are unicast and therefore cancelable.

See the accompanying sketch of a task implementation.

There is also an esoteric difference between a promise and a future. Promise resolvers accept either a value or a promise and will recursively unwrap transitive promises for promises. In most if not all strongly typed languages, this behavior makes it hard if not impossible to infer the type of a promise. A future is a promise's strongly typed alter ego, which can take advantage of type inference to avoid having to explicitly cast the type of a promise.

Promises, tasks, and futures are three variations of a singular reactive value. They vary by being either broadcast or unicast, or by being suitable for strong or weak type systems.

6.3.3. Plural and temporal

There are many plural reactive value types. Each has a different adaptation for dealing with limited resources.

A stream has many of the same constraints as an array. Imagine a plane with space and time. If you rotate an array from the space axis to the time axis, it would become a stream. The order is important, and every value is significant.

Consumers and producers are unlikely to process values at the same rate. If the consumer is faster than the producer, it must idle between receiving values. If a producer is faster than their corresponding consumer, it must idle between sending values.

This pushing and pulling is captured by the concept of pressure. On the producer side, a vacuum stalls the consumer and a pressure sends values forward. On the consumer side, a vacuum draws values forward and pressure, often called back pressure, stalls the producer. Pressure exists to ensure that every value transits the setter to the getter.

Since the consumer of a stream expects to see every value, streams are unicast like tasks. As they are unicast they are also cancelable. Streams are a cooperation

between the reader and the writer and information flows both ways. Data flows forward, acknowledgements flow backward, and either the consumer or producer can terminate the flow.

Although a stream is unicast, it is certainly possible to branch a stream into multiple streams in a variety of ways. A fork in a stream is an operator that ensures that every value gets sent to each of an array of consumers. The slowest of the forks determines the pressure, so the pressure of a fork can only be higher than that of a single consumer. The simpler strategy of providing a stream to multiple consumers produces a "round robin" load balancing effect, where each consumer receives an exclusive, possibly random, portion of the stream. The pressure of a shared stream can only be lower than that of a single consumer.

In the following example, the map operator creates two new streams from a single input stream. The slow map will see half as many values as the fast map. The slow map will consume and produce five values per second, and the fast map will consume and produce ten, sustaining a maximum throughput of fifteen values per second if the original stream can produce values that quickly. If the original stream can only produce ten or less values per second, the values will be distributed fairly between both consumers. In contrast, publishers and subscribers are broadcast. Information flows only one direction, from the publishers to the subscribers. Also, there is no guarantee of continuity. The publisher does not wait for a subscriber and the subscriber receives whatever values were published during the period of their subscription. A stream would buffer all values produced until the consumer arrives.

With time series data, values that change over time but belie the same meaning, order and integrity may not be important. For example, if you were bombarded with weather forecasts, you could discard every report except the one you most recently received. Alternately, consider a value that represents the current time. Since the current time is always changing, it would not be meaningful, much less possible, to respond every moment it changes.

Time series data comes in two varieties: discrete and continuous. Discrete values should be pushed whereas continuous values should be pulled or polled. (If a homophone is a disaster, what are synonymous homophones?)

The current time or temperature are examples of continuous behaviors. Animation frames and morse code are examples of discrete signals.

6.3.4. Primitives

Let us consider each primitive in detail. Since the temporal primitives have spatial analogies, and since some of these spatial primitives are relatively new to JavaScript, we will review these first.

6.3.4.1. Iterators

An iterator is an object that allows us to lazily but synchronously consume multiple values. Iterators are not new to JavaScript, but there is a new standard forming at time of writing.

Iterators implement a next() method that returns an object that may have a value property, and may have a done property. Although the standard does not give this object a name, we will call it an iteration. If the iterator has produced the entirety of a sequence, the done property of the iteration will be true. Generator functions return iterators that expand on this basic definition. The value of a non-final iteration corresponds to a yield expression and the value of a done iteration corresponds to a return expression.

Iterators are an interface with many implementations. The canonical iterator yields the values from an array.

What distinguishes an iterator from an array is that it is lazy. An iterator does not necessarily end. We can have iterators for non-terminating sequences, like counting or the fibonacci sequence. The range function produces a sequence of values within an interval and separated by a stride.

If the stop value of the range is Infinity, the iterator will have no end, and will never produce a done iteration. Unlike an array, an indefinite iterator consumes no more memory than an empty one.

The eager equivalent would produce an array, but would only work for bounded intervals since it must create an exhaustive collection in memory before returning.

Iterators may have alternate implementations of some methods familiar from arrays. For example, forEach would walk the iterator until it is exhausted. map would produce a new iterator of values passed through some transform, while filter would produce a new iterator of the values that pass a test. An iterator

can support reduce, which would exhaust the iteration, but reduceRight would be less sensible since iterators only walk forward. Iterators may also have some methods that are unique to their character, like dropWhile and takeWhile.

We can save time and space by implementing pipelines with iterators instead of arrays. The following example can be interpreted as either eager or lazy, depending on whether range returns an array or an iterator. If we start with an array, map will create another array of 1000 values and filter will create another large array. If we start with an iterator, we will never construct an array of any size, instead percolating one value at a time as the reducer pulls them from the filter, as the filter pulls them from the mapping, and as the mapping pulls them from the range.

6.3.4.2. Generator Functions

Consider the eager and lazy range function implementations. We lose a certain clarity when we convert the array range maker into an iterator range maker. Generator functions alleviate this problem by offering a way to express iterations procedurally, with a lazy behavior.

A JavaScript engine near you may already support generator functions. The syntax consists of adding an asterisk to the function declaration and using yield to produce iterations. Calling a generator function does not execute the function, but instead sets up a state machine to track where we are in the function and returns an iterator. Whenever we ask the iterator for an iteration, the state machine will resume the execution of the function until it produces an iteration or terminates.

Notice that the range generator function restores and perhaps even exceeds the clarity of the range array maker.

Calling next has three possible outcomes. If the iterator encounters a yield, the iteration will have a value. If the iterator runs the function to either an express or implied return, the iteration will have a value and done will be true. If the iterator runs to an explicit return, this terminal iteration carries the return value. If the generator function throws an error, this will propagate out of next().

Generators and iterators are unicast. The consumer expects to see every value

from the producer. Since generators and iterators cooperate, information flows both forward as values, and backward as requests for more values.

However, the consumer can send other information back to the producer. The next method, familiar from basic iterators, gains the ability to determine the value of the yield expression from which the generator resumes. As a trivial example, consider a generator that echoes whatever the consumer requests.

We must prime the generator because it does not begin with a yield. We advance the state machine to the first yield and allow it to produce the initial, undefined message. We then populate the message variable with a value, receiving its former undefined content again. Then we begin to see the fruit of our labor as the values we previously sent backward come forward again. This foreshadows the ability of stream readers to push back on stream writers.

Additionally, the iterator gains a throw method that allows the iterator to terminate the generator by causing the yield expression to raise the given error. The error will unravel the stack inside the generator. If the error unravels a try-catch-finally block, the catch block may handle the error, leaving the generator in a resumable state if the returned iteration is not done. If the error unravels all the way out of the generator, it will pass into the stack of the throw caller.

The iterator also gains a return method that causes the generator to resume as if from a return statement, regardless of whether it actually paused at a yield expression. Like a thrown error, this unravels the stack, executing finally blocks, but not catch blocks.

As such, like next, the throw and return methods may either return an iteration, done or not, or throw an error. This foreshadows the ability of a stream reader to prematurely stop a stream writer.

iterator.throw(new Error("Do not want!")); Note that in Java, iterators have a hasNext() method. This is not implementable for generators owing to the Halting Problem. The iterator must try to get a value from the generator before the generator can conclude that it cannot produce another value.

6.3.4.3. Generators

There is no proposal for a standard generator, but for the sake of completeness, if an array iterator consumes an array, an array generator would lazily produce

one. An array generator object would implement yield as a method with behavior analogous to the same keyword within a generator function. The yield method would add a value to the array.

Since ECMAScript 5, at Doug Crockford's behest, JavaScript allows keywords to be used for property names, making this parallel between keywords and methods possible. A generator might also implement return and throw methods, but a meaningful implementation for an array generator is a stretch of the imagination. Although an array generator is of dubious utility, it foreshadows the interface of asynchronous generators, for which meaningful implementations of return and throw methods are easier to obtain, and go on to inform a sensible design for asynchronous generator functions.

6.3.4.4. Asynchronous Values

The asynchronous analogue of a getter is a promise. Each promise has a corresponding resolver as its asynchronous setter. Collectively the promise and resolver are a deferred value.

The salient method of a promise is then, which creates a new promise for the result of a function that will eventually observe the value of the promise. If a promise were plural, the then method might be called map. If you care to beg an esoteric distinction, it might be called map if the observer returns a value and flatMap if the observer returns a promise. The then method of a promise allows either.

Promises can also have a done method that observes the value but does not return a promise nor captures the result of the observer. Again, if a promise were plural, the done method might be called forEach.

The then method also supports a second function that would observe whether the input promise radiates an exception, and there is a catch method to use as a shorthand if you are only interested in catching errors.

At this point, the design described here begins to differ from the standard Promise proposed for ECMAScript 6, arriving in browsers at time of writing. The purpose of these differences is not to propose an alternative syntax, but to reinforce the relationship between a promise and its conceptual neighbors.

A resolver is the singular analogue of a generator. Rather than yielding, returning, and throwing errors, the resolver can only return or throw.

With the standard promise, a free resolve function is sufficient and ergonomic for expressing both of these methods. resolver.return(promise) is equivalent to resolve(promise). resolver.return(10) is equivalent to resolve(10) or resolve(Promise.resolve(10))sin non-promise values are automatically boxed in an already-fulfilled promise. resolver.throw(error) is equivalent to resolve(Promise.reject(error)). In all positions, resolve is the temporal analogue of return and reject is the temporal analogue of throw. Since promises as we know them today bridged the migration gap from ECMAScript 3 to ECMAScript 6, it was also necessary to use non-keywords for method names.

A deferred value can be deferred further by resolving it with another promise. This can occur either expressly through the resolver, or implicitly by returning a promise as the result of a observer function.

The then method internally creates a new deferred, returns the promise, and later forwards the return value of the observer to the resolver. This is a sketch of a then method that illustrates this adapter. Note that we create a deferred, use the resolver, and return the promise. The adapter is responsible for catching errors and giving the consumer an opportunity to do further work or to recover.

The standard Promise does not reveal Promise.defer(). Instead, it is hidden by then and by the Promise constructor, which elects to hide the deferred object and the resolver object, instead "revealing" the resolve and reject methods as free arguments to a setup function, furthering the need to give these functions names that are not keywords.

With a promise, information flows only from the first call to a resolver method to all promise observers, whether they are registered before or after the resolution.

With a task, information flows from the first call to a resolver method to the first call to an observer method, regardless of their relative order, but one kind of information can flow upstream. The observer may unsubscribe with an error. This is conceptually similar to throwing an error back into a generator from an iterator and warrants the same interface.

This interface foreshadows its plural analogue: streams.

6.3.4.5. Asynchronous Functions

Generator functions have existed in other languages, like Python, for quite some time, so folks have made some clever uses of them. We can combine promises and generator functions to emulate asynchronous functions. The key insight is a single, concise method that decorates a generator, creating an internal "promise trampoline". An asynchronous function returns a promise for the eventual return value, or the eventual thrown error, of the generator function. However, the function may yield promises to wait for intermediate values on its way to completion. The trampoline takes advantage of the ability of an iterator to send a value from next to yield.

Mark Miller's implementation of the async decorator is succinct and insightful. We produce a wrapped function that will create a promise generator and proceed immediately. Each requested iteration has three possible outcomes: yield, return, or throw. Yield waits for the given promise and resumes the generator with the eventual value. Return stops the trampoline and returns the value, all the way out to the promise returned by the async function. If you yield a promise that eventually throws an error, the async function resumes the generator with that error, giving it a chance to recover.

As much as JavaScript legitimizes the async promise generators by supporting returning and throwing, now that Promises are part of the standard, the powers that sit on the ECMAScript standards committee are contemplating special async and await syntax for this case. The syntax is inspired by the same feature of C#.

One compelling reason to support special syntax is that await may have higher precedence than the yield keyword.

By decoupling async functions from generator functions, JavaScript opens the door for async generator functions, foreshadowing a plural and temporal getter, a standard form for readable streams.

6.3.4.6. Promise Queues

Consider an asynchronous queue. With a conventional queue, you must put a value in before you can take it out. That is not the case for a promise queue. Just as you can attach an observer to a promise before it is resolved, with a promise

queue, you can get a promise for the next value in order before that value has been given.

Likewise of course you can add a value to the queue before observing it.

Although promises come out of the queue in the same order their corresponding resolutions enter, a promise obtained later may settle sooner than another promise. The values you put in the queue may themselves be promises.

A promise queue qualifies as an asynchronous collection, specifically a collection of results: values or thrown errors captured by promises. The queue is not particular about what those values mean and is a suitable primitive for many more interesting tools.

Interface			
PromiseQueue	Value	Plural	Temporal
queue.get	Getter	Plural	Temporal
queue.put	Setter	Plural	Temporal

The implementation of a promise queue is sufficiently succinct that there's no harm in embedding it here. This comes from Mark Miller's Concurrency Strawman for ECMAScript and is a part of the Q library, exported by the q/queue module. Internally, a promise queue is an asynchronous linked list that tracks the head promise and tail deferred. get advances the head promise and put advances the tail deferred.

The implementation uses methods defined in a closure. Regardless of how this is accomplished, it is important that it be possible to pass the free get function to a consumer and put to a producer to preserve the principle of least authority and the unidirectional flow of data from producer to consumer.

See the accompanying sketch of a promise queue implementation.

A promise queue does not have a notion of termination, graceful or otherwise. We will later use a pair of promise queues to transport iterations between streams.

6.3.4.7. Semaphores

Semaphores are flags or signs used for communication and were precursors to telegraphs and traffic lights. In programming, semaphores are usually used to synchronize programs that share resources, where only one process can use a

resource at one time. For example, if a process has a pool of four database connections, it would use a semaphore to manage that pool.

Typically, semaphores are used to block a thread or process from continuing until a resource becomes available. The process will "down" the semaphore whenever it enters a region where it needs a resource, and will "up" the semaphore whenever it exits that region. The terminology goes back to raising and lowering flags. You can imagine your process as being a train and a semaphore as guarding the entrance to a particular length of track. Your process stops at the gate until the semaphore goes up.

Of course, in a reactive program, we don't block. Instead of blocking, we return promises and continue when a promise resolves. We can use a promise as a non-blocking mutex for a single resource, and a promise queue as a non-blocking semaphore for multiple resources.

In this example, we establish three database connections that are shared by a function that can be called to do some work with the first available connection. We get the resource, do our work, and regardless of whether we succeed or fail, we put the resource back in the pool.

6.3.4.8. Promise Buffers

Consider another application. You have a producer and a consumer, both doing work asynchronously, the producer periodically sending a value to the consumer. To ensure that the producer does not produce faster than the consumer can consume, we put an object between them that regulates their flow rate: a buffer. The buffer uses a promise queue to transport values from the producer to the consumer, and another promise queue to communicate that the consumer is ready for another value from the producer. The following is a sketch to that effect.

This sketch uses the vernacular of iterators and generators, but each of these has equivalent nomenclature in the world of streams.

in.yield means "write". in.return means "close". in.throw means "terminate prematurely with an error". out.next means "read". out.throw means "abort or cancel with an error". out.return means "abort or cancel prematurely but without an error". So a buffer fits in the realm of reactive interfaces. A buffer

has an asynchronous iterator, which serves as the getter side. It also has an asynchronous generator, which serves as the setter dual. The buffer itself is akin to an asynchronous, plural value. In addition to satisfying the requirements needed just to satisfy the triangulation between synchronous iterables and asynchronous promises, it solves the very real world need for streams that support pressure to regulate the rate of flow and avoid over-commitment. An asynchronous iterator is a readable stream. An asynchronous generator is a writable stream.

Stream				
Promise	Buffer	Value	Plural	Temporal
Promise	Iterator	Getter	Plural	Temporal
Promise	Generator	Setter	Plural	Temporal

A buffer has a reader and writer, but there are implementations of reader and writer that interface with the outside world, mostly files and sockets.

In the particular case of an object stream, if we treat yield and next as synonyms, the input and output implementations are perfectly symmetric. This allows a single constructor to serve as both reader and writer. Also, standard promises use the Revealing Constructor pattern, exposing the constructor for the getter side. The standard hides the Promise.defer() constructor method behind the scenes, only exposing the resolver as arguments to a setup function, and never revealing the promise, resolver deferred object at all. Similarly, we can hide the promise buffer constructor and reveal the input side of a stream only as arguments to the output stream constructor.

The analogous method to Promise.defer() might be Stream.buffer(), which would return an in, out pair of entangled streams.

See the accompanying sketch of a stream implementation.

6.3.4.9. Promise Iterators

One very important kind of promise iterator lifts a spatial iterator into the temporal dimension so it can be consumed on demand over time. In this sketch, we just convert a synchronous next method to a method that returns a promise for the next iteration instead. We use a mythical iterate function which would create iterators for all sensible JavaScript objects and delegate to the iterate method of

anything else that implements it. There is talk of using symbols in ES7 to avoid recognizing accidental iterables as this new type of duck.

The conversion may seem superfluous at first. However, consider that a synchronous iterator might, apart from implementing next(), also implement methods analogous to Array, like forEach, map, filter, and reduce. Likewise, an asynchronous iterator might provide analogues to these functions lifted into the asynchronous realm.

The accompanying sketch of a stream constructor implements a method Stream.from, analogous to ECMAScript 6's own Array.from. This function coerces any iterable into a stream, consuming that iterator on demand. This allows us, for example, to run an indefinite sequence of jobs, counting from 1, doing four jobs at any time.

map

For example, asynchronous map would consume iterations and run jobs on each of those iterations using the callback. However, unlike a synchronous map, the callback might return a promise for its eventual result. The results of each job would be pushed to an output reader, resulting in another promise that the result has been consumed. This promise not only serves to produce the corresponding output iteration, but also serves as a signal that the job has completed, that the output has been consumed, and that the map can schedule additional work. An asynchronous map would accept an additional argument that would limit the number of concurrent jobs.

forEach

Synchronous forEach does not produce an output collection or iterator. However, it does return undefined when it is done. Of course synchronous functions are implicitly completed when they return, but asynchronous functions are done when the asynchronous value they return settles. forEach returns a promise for undefined.

Since streams are unicast, asynchronous forEach would return a task. It stands to reason that the asynchronous result of forEach on a stream would be able to propagate a cancellation upstream, stopping the flow of data from the producer side. Of course, the task can be easily forked or coerced into a promise if it needs to be shared freely among multiple consumers.

Like map it would execute a job for each iteration, but by default it would perform these jobs in serial. Asynchronous forEach would also accept an additional argument that would expand the number of concurrent jobs. In this example, we would reach out to the database for 10 user records at any given time.

reduce

Asynchronous reduce would aggregate values from the input reader, returning a promise for the composite value. The reducer would have an internal pool of aggregated values. When the input is exhausted and only one value remains in that pool, the reducer would resolve the result promise. If you provide a basis value as an argument, this would be used to "prime the pump". The reducer would then start some number of concurrent aggregator jobs, each consuming two values. The first value would preferably come from the pool, but if the pool is empty, would come from the input. The second value would come unconditionally from the input. Whenever a job completes, the result would be placed back in the pool.

pipe

An asynchronous iterator would have additional methods like copy or pipe that would send iterations from this reader to another writer. This method would be equivalent to using forEach to forward iterations and then to terminate the sequence.

iterator.copy(generator); // is equivalent to: iterator.forEach(generator.yield).then(generator.re
generator.throw); Note that the promise returned by yield applies pressure on the forEach machine, pushing ultimately back on the iterator.

buffer

It would have buffer which would construct a buffer with some capacity. The buffer would try to always have a value on hand for its consumer by prefetching values from its producer. If the producer is faster than the consumer, this can help avoid round trip latency when the consumer needs a value from the producer.

read

Just as it is useful to transform a synchronous collection into an iterator and an iterator into a reader, it is also useful to go the other way. An asynchronous iterator would also have methods that would return a promise for a collection

of all the values from the source, for example all, or in the case of readers that iterate collections of bytes or characters, join or read.

Remote iterators

Consider also that a reader may be a proxy for a remote reader. A promise iterator be easily backed by a promise for a remote object.

Apart from then and done, promises provide methods like get, call, and invoke to allow promises to be created from promises and for messages to be pipelined to remote objects. An iterate method should be a part of that protocol to allow values to be streamed on demand over any message channel.

6.3.4.10. Promise Generators

A promise generator is analogous in all ways to a plain generator. Promise generators implement yield, return, and throw. The return and throw methods both terminate the stream. Yield accepts a value. They all return promises for an acknowledgement iteration from the consumer. Waiting for this promise to settle causes the producer to idle long enough for the consumer to process the data.

One can increase the number of promises that can be held in flight by a promise buffer. The buffer constructor takes a length argument that primes the acknowledgement queue, allowing you to send that number of values before having to wait for the consumer to flush.

If the consumer would like to terminate the producer prematurely, it calls the throw method on the corresponding promise iterator. This will eventually propagate back to the promise returned by the generator's yield, return, or throw.

6.3.4.11. Asynchronous Generator Functions

Jafar Husain recently asked the ECMAScript committee, whether generator functions and async functions were composable, and if so, how they should compose. (His proposal continues to evolve.)

One key question is what type an async generator function would return. We look to precedent. A generator function returns an iterator. A asynchronous function

returns a promise. Should the asynchronous generator return a promise for an iterator, an iterator for promises?

If Iterator¡T¿ means that an iterator implements next such that it produces Iteration¡T¿, the next method of an Iterator¡Promise¡T¿¿ would return an Iteration¡Promise¡T¿¿, which is to say, iterations that carry promises for values.

There is another possibility. An asynchronous iterator might implement next such that it produces Promise¡Iteration¡T¿¿ rather than Iteration¡Promise¡T¿¿. That is to say, a promise that would eventually produce an iteration containing a value, rather than an iteration that contains a promise for a value.

This is, an iterator of promises, yielding Iteration¡Promise¡T¿¿:

This is a promise iterator, yielding Promise¡Iteration¡T¿¿:

Promises capture asynchronous results. That is, they capture both the value and error cases. If next returns a promise, the error case would model abnormal termination of a sequence. Iterations capture the normal continuation or termination of a sequence. If the value of an iteration were a promise, the error case would capture inability to transport a single value but would not imply termination of the sequence.

In the context of this framework, the answer is clear. An asynchronous generator function uses both await and yield. The await term allows the function to idle until some asynchronous work has settled, and the yield allows the function to produce a value. An asynchronous generator returns a promise iterator, the output side of a stream.

Recall that an iterator returns an iteration, but a promise iterator returns a promise for an iteration, and also a promise generator returns a similar promise for the acknowledgement from the iterator.

The following example will fetch quotes from the works of Shakespeare, retrieve quotes from each work, and push those quotes out to the consumer. Note that the yield expression returns a promise for the value to flush, so awaiting on that promise allows the generator to pause until the consumer catches up.

It is useful for await and yield to be completely orthogonal because there are cases where one will want to yield but ignore pressure from the consumer, forcing the iteration to buffer.

Jafar also proposes the existence of an on operator. In the context of this framework, the on operator would be similar to the ECMAScript 6 of operator, which accepts an iterable, produces an iterator, and then walks the iterator.

The on operator would operate on an asynchronous iterable, producing an asynchronous iterator, and await each promised iteration. Look for the await in the following example.

One point of interest is that the on operator would work for both asynchronous and synchronous iterators and iterables, since await accepts both values and promises.

Jafar proposes that the asynchronous analogues of iterate() would be observe(generator), from which it is trivial to derrive forEach, but I propose that the asynchronous analogues of iterate() would just be iterate() and differ only in the type of the returned iterator. What Jafar proposes as the asyncIterator.observe(asyncGenerator) method is effectively equivalent to synchronous iterator.copy(generator) or stream.pipe(stream). In this framework, copy would be implemented in terms of forEach.

And, forEach would be implemented in terms of next, just as it would be layered on a synchronous iterator.

6.3.4.12. Observables

There is more than one way to solve the problem of processor contention or process over-scheduling. Streams have a very specific contract that makes pressurization necessary. Specifically, a stream is intended to transport the entirety of a collection and strongly resembles a spatial collection that has been rotated 90 degrees onto the temporal axis. However, there are other contracts that lead us to very different strategies to avoid over-commitment and they depend entirely on the meaning of the data in transit. The appropriate transport is domain specific.

Consider a sensor, like a thermometer or thermocouple. At any given time, the subject will have a particular temperature. The temperature may change continuously in response to events that are not systematically observable. Suppose that you poll the thermocouple at one second intervals and place that on some plural, asynchronous setter. Suppose that this ultimately gets consumed by a visualization that polls the corresponding plural, asynchronous getter sixty times

per second. The visualization is only interested in the most recently sampled value from the sensor.

Consider a variable like the position of a scrollbar. The value is discrete. It does not change continuously. Rather, it changes only in response to an observable event. Each time one of these scroll events occurs, we place the position on the setter side of some temporal collection. Any number of consumers can subscribe to the getter side and it will push a notification their way.

However, if we infer a smooth animation from the discrete scroll positions and their times, we can sample the scroll position function on each animation frame.

These cases are distinct from streams and have interesting relationships with each other. With the temperature sensor, changes are continuous, whereas with the scroll position observer, the changes are discrete. With the temperature sensor, we sample the data at a much lower frequency than the display, in which case it is sufficient to remember the last sensed temperature and redisplay it. If we were to sample the data at a higher frequency than the display, it would be sufficient for the transport to forget old values each time it receives a new one. Also, unlike a stream, these cases are both well adapted for multiple-producer and multiple-consumer scenarios.

Also unlike streams, one of these concepts pushes data and the other polls or pulls data. A stream has pressure, which is a kind of combined pushing and pulling. Data is pulled toward the consumer by a vacuum. Producers are pushed back by pressure when the vacuum is filled, thus the term: back-pressure.

The discrete event pusher is a Signal. The continuous, pollable is a Behavior.

Interface		
Signal Observable	Get	Push
Signal Generator	Set	Push
Signal Value	Push	
Behavior Iterator	Get	Poll
Behavior Generator	Set	Poll
Behavior Value	Poll	

6.3.4.13. Observables and Signals

A signal represents a value that changes over time. The signal is asynchronous and plural, like a stream. Unlike a stream, a signal can have multiple producers and consumers. The output side of a signal is an observable.

A signal has a getter side and a setter side. The asynchronous getter for a signal is an observable instead of a reader. The observable implements forEach, which subscribes an observer to receive push notifications whenever the signal value changes.

The signal generator is the asynchronous setter. Like a stream writer, it implements yield. However, unlike a stream writer, yield does not return a promise.

signal.in.yield(10); Signals do not support pressure. Just as yield does not return a promise, the callback you give to forEach does not accept a promise. A signal can only push. The consumer (or consumers) cannot push back.

Observables also implement next, which returns an iteration that captures the most recently dispatched value. This allows us to poll a signal as if it were a behavior.

See the accompanying sketch of a observable implementation.

Just as streams relate to buffers, not every observable must be paired with a signal generator. A noteworthy example of an external observable is a clock. A clock emits a signal with the current time at a regular period and offset.

See the accompanying sketch of a clock implementation.

Signals may correspond to system or platform signals like keyboard or mouse input or other external sensors. Furthermore, a signal generator might dispatch a system level signal to another process, for example SIGHUP, which typically asks a daemon to reload its configuration.

daemon.signals.yield(”SIGHUP”);

6.3.4.14. Behaviors

A behavior represents a time series value. A behavior may produce a different value for every moment in time. As such, they must be polled at an interval meaningful to the consumer, since the behavior itself has no inherent resolution.

Behaviors are analogous to Observables, but there is no corresponding setter, since it produces values on demand. The behavior constructor accepts a function that returns the value for a given time. An asynchronous behavior returns promises instead of values.

See the accompanying sketch of a behavior implementation.

6.3.4.15. Cases

Progress and estimated time to completion

Imagine you are copying the values from a stream into an array. You know how long the array will be and when you started reading. Knowing these variables and assuming that the rate of flow is steady, you can infer the amount of progress that has been made up to the current time. This is a simple matter of dividing the number of values you have so far received, by the total number of values you expect to receive.

var progress = index / length; This is a discrete measurement that you can push each time you receive another value. It is discrete because it does not change between events.

You can also infer the average throughput of the stream, also a discrete time series.

var elapsed = now - start; var throughput = index / elapsed; From progress you can divine an estimated time of completion. This will be the time you started plus the time you expect the whole stream to take.

var stop = start + elapsed / progress; var stop = start + elapsed / (index / length); var stop = start + elapsed * length / index; We could update a progress bar whenever we receive a new value, but frequently we would want to display a smooth animation continuously changing. Ideally, progress would proceed linearly from 0 at the start time to 1 at the stop time. You could sample progress at any moment in time and receive a different value. Values that lack an inherent resolution are continuous. It becomes the responsibility of the consumer to determine when to sample, pull or poll the value.

For the purposes of a smooth animation of a continuous behavior, the frame rate is a sensible polling frequency. We can infer a continuous progress time series

from the last known estimated time of completion.

var progress = (now - start) / (estimate - start);

6.3.5. Summary

Reactive primitives can be categorized in multiple dimensions. The interfaces of analogous non-reactive constructs including getters, setters, and generators are insightful in the design of their asynchronous counterparts. Identifying whether a primitive is singular or plural also greatly informs the design.

We can use pressure to deal with resource contention while guaranteeing consistency. We can alternately use push or poll strategies to skip irrelevant states for either continuous or discrete time series data with behaviors or signals.

There is a tension between cancelability and robustness, but we have primitives that are useful for both cases. Streams and tasks are inherently cooperative, cancelable, and allow bidirectional information flow. Promises guarantee that consumers and producers cannot interfere.

All of these concepts are related and their implementations benefit from mutual availability. Promises and tasks are great for single result data, but can provide a convenient channel for plural signals and behaviors.

Bringing all of these reactive concepts into a single framework gives us an opportunity to tell a coherent story about reactive programming, promotes a better understanding about what tool is right for the job, and obviates the debate over whether any single primitive is a silver bullet.

6.3.6. Further Work

There are many more topics that warrant discussion and I will expand upon these here.

Reservoir sampling can be modeled as a behavior that watches a stream or signal and produces a representative sample on demand.

A clock user interface is a good study in the interplay between behaviors, signals, time, and animation scheduling.

Drawing from my experience at FastSoft, we exposed variables from the kernel's networking stack so we could monitor the bandwidth running through our TCP acceleration appliance. Some of those variables modeled the number of packets transmitted and the number of bytes transmitted. These counters would frequently overflow. There are several interesting ways to architect a solution that would provide historical data in multiple resolutions, accounting for the variable overflow, involving a combination of streams, behaviors, and signals. I should draw your attention to design aspects of RRDTool.

An advantage of having a unified framework for reactive primitives is to create simple stories for passing one kind of primitive to another. Promises can be coerced to tasks, tasks to promises. A signal can be used as a behavior, and a behavior can be captured by a signal. Signals can be channeled into streams, and streams can be channeled into signals.

It is worth exploring in detail how operators can be lifted in each of these value spaces.

Implementing distributed sort using streams is also a worthy exercise.

Asynchronous behaviors would benefit from an operator that solves the thundering herd problem, the inverse of throttling.

How to implement type ahead suggestion is a great case to explore cancelable streams and tasks.

I also need to discuss how these reactive concepts can propagate operational transforms through queries, using both push and pull styles, and how this relates to bindings, both synchronous and asynchronous.

I also need to compare and contrast publishers and subscribers to the related concepts of signals and streams. In short, publishers and subscribers are broadcast, unlike unicast streams, but a single subscription could be modeled as a stream. However, a subscriber can typically not push back on a publisher, so how resource contention is alleviated is an open question.

Related to publishing and subscribing, streams can certainly be forked, in which case both branches would put pressure back on the source.

Streams also have methods that return tasks. All of these could propagate estimated time to completion. Each of the cases for all, any, race, and read are worth exploring.

High performance buffers for bytewise data with the promise buffer interface require further exploration.

Implementing a retry loop with promises and tasks is illustrative.

Reactive Extensions (Rx) beg a distinction between hot and cold observables, which is worth exploring. The clock reference implementation shows one way to implement a signal that can be active or inactive based on whether anyone is looking.

The research into continuous behaviors and the original idea of Functional Reactive Programming by Conal Elliott deserves attention.

The interplay between promises and tasks with their underlying progress behavior and estimated time to completion and status signals require further explanation. These ideas need to be incorporated into the sketches of promise and task implementations.

6.4. Approaches for asynchronous information flow handling

- *Continuation Passing.* Definition and call of a function within function which performs asynchronous operation
- *Callbacks.* Passing as last input parameter function with specific signature for iinput paramteres (error, data), which performs operation/error handling over returned data after it was obtained.
- *Async.* JavaScript library which gives an opportunity to write asynchronous functions execution in chain or parallel form within single information flow.
- *Promises.* Function can return promise object with can be resolved or rejected in a later time when it particular method will be called by other function.
- *Streams.* Piping asynchronous functions one to another. Result of first will be pushed to next stream whenever it is available, or pulled by the next stream whenever it is necessary (depends from stream mode)

- *Generators.* Type of constructor introduced in ECMA6 which returns iterator value. Which will iterate through the code unless it will rich yeild method which will wait for return value without blocking event loop.

"The original and flattened solutions are the fastest, as they use vanilla callbacks, with the fastest flattened solution being flattened-class.js."[2]

6.5. Stream

"A stream is an abstract interface implemented by various objects in Node.js." [18] "Dominic Tarr (one of top contributors to the Node.js community [22]), defines streams as node's best and most misunderstood idea."[33]

Streams are the classic example of Pipe-and-filter architecture.

"In an event-based platform such as Node.js, the most efficient way to handle I/O is in real time, consuming the input as soon as it is available and sending the output as soon as it is produced by the application."[33]

Spatial efficiency. "First of all, streams allow us to do things that would not be possible, by buffering data and processing it all at once. For example, consider the case in which we have to read a very big file, let's say, in the order of hundreds of megabytes or even gigabytes. Clearly, using an API that returns a big buffer when the file is completely read, is not a good idea. Imagine reading a few of these big files concurrently; our application will easily run out of memory. Besides that, buffers in V8 (default NodeJS engine) cannot be bigger than 0x3FFFFFFF bytes (a little bit less than 1 GB). So, we might hit a wall way before running out of physical memory." [33]

Time efficiency. "Let's now consider the case of an application that compresses a file and uploads it to a remote HTTP server, which in turn decompresses and saves it on the filesystem. If our client was implemented using a buffered API, the upload would start only when the entire file has been read and compressed. On the other hand, the decompression will start on the server only when all the data has been received. A better solution to achieve the same result involves the use of streams. On the client machine, streams allows you to compress and send

the data chunks as soon as they are read from the filesystem, whereas, on the server, it allows you to decompress every chunk as soon as it is received from the remote peer."[33]

Composability. "The code we have seen so far has already given us an overview of how streams can be composed, thanks to the pipe() method, which allows us to connect the different processing units, each being responsible for one single functionality in perfect Node.js style. This is possible because streams have a uniform interface, and they can understand each other in terms of API. The only prerequisite is that the next stream in the pipeline has to support the data type produced by the previous stream, which can be either binary, text, or even objects, as we will see later in the chapter.
For these reasons, streams are often used not just to deal with pure I/O, but also as a means to simplify and modularize the code."[33]

6.5.1. Anatomy of Streams

"Every stream in Node.js is an implementation of one of the four base abstract classes available in the stream core module:

- stream.Readable
- stream.Writable
- stream.Duplex
- stream.Transform

Each stream class is also an instance of EventEmitter. Streams, in fact, can produce several types of events, such as end, when a Readable stream has finished reading, or error, when something goes wrong.

One of the reasons why streams are so flexible is the fact that they can handle not only binary data, but practically, almost any JavaScript value; in fact they can support two operating modes:

- Binary mode: This mode is where data is streamed in the form of chunks, such as buffers or strings;
- Object mode: This mode is where the streaming data is treated as a sequence of discreet objects (allowing to use almost any JavaScript value).

Readable streams. A readable stream represents a source of data; in Node.js, it's implemented using the Readable abstract class that is available in the stream module.

Writable streams. A writ[e]able stream represents a data destination; in Node.js, it's implemented using the Writ[e]able abstract class, which is available in the stream module.

Duplex streams. A Duplex stream is a stream that is both Readable and Writ[e]able. It is useful when we want to describe an entity that is both a data source and a data destination, as for example, network sockets. Duplex streams inherit the methods of both stream.Readable and stream.Writable, so this is nothing new to us. This means that we can read() or write() data, or listen for both the readable and drain events.

Transform streams. The Transform streams are a special kind of Duplex stream that are specifically designed to handle data transformations. In a simple Duplex stream, there is no immediate relationship between the data read from the stream and the data written into it (at least, the stream is agnostic to such a relationship). On the other side, Transform streams apply some kind of transformation to each chunk of data that they receive from their Writable side and then make the transformed data available on their Readable side. From the outside, the interface of a Transform stream is exactly like that of a Duplex stream. However, when we want to build a new Duplex stream we have to provide the read() and write() methods while, for implementing a new Transform stream, we have to fill in another pair of methods: transform() and flush()."[33]

6.5.2. Piping patterns

"As in real-life plumbing, Node.js streams also can be piped together following different patterns; we can, in fact, merge the flow of two different streams into one, split the flow of one stream into two or more pipes, or redirect the flow based on a condition. In this section, we are going to explore the most important plumbing techniques that can be applied to Node.js streams."[33]

Combining streams - 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.[33]

Forking streams - piping single readable in to multiple writable streams.[33]

Merging streams - piping multiple readable streams in to single writable stream.[33]

Multiplexing and demultiplexing - forking and merging pattern which provides shared communication chanel for entities from different sreams.[33] [33]

6.6. Performance measurements by Gorgi Kosev

6.7. Asynchronous Programming

here is the stuff from General Theory of Reactivity

7. OOP and FP

7.1. Object Oriented Programming

"Programming languages with objects and classes typically provide dynamic lookup, abstraction, subtyping, and inheritance. These are the four main language concepts for object-oriented programming. They may be summarized in the following manner: *Dynamic lookup* means that when a message is sent to an object, the function code (or method) to be executed is determined by the way that the object is implemented, not some static property of the pointer or variable used to name the object. In other words, the object "chooses" how to respond to a message, and different objects may respond to the same message in different ways. *Abstraction* means that implementation details are hidden inside a program unit with a specific interface. For objects, the interface usually consists of a set of public functions (or public methods) that manipulate hidden data. *Subtyping* means that if some object a has all of the functionality of another object b, then we may use a in any context expecting b. *Inheritance* is the ability to reuse the definition of one kind of object to define another kind of object. "[43]

7.1.1. Design Principles

Agile design is a process, not an event. It's the continous application of principles, patterns, and practices to improve the structure and readability of the software. It is the dedication to keep the design of the system as simple, clean, and expressive as possible at all times.[42]

Symptoms of not agile design [42]:

- 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;

Foundamental Object Oriented design principles[35][42] :

- Closing - Encapsulate things in your design that are likely to change.
- Code to an Interface - rather then to an implementation.
- Do not repeat yourself (DRY) - Avoid duplicate code.
- The Single-Responsibility Principle (SRP) - A class should have only one reason to change
- The Open/Closed Principle (OCP) - Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification
- The Liskov Substitution Principle - Subtypes must be substitutable for their base types.
- The Dependency-Inversion Principle - A) High-level modules should not depend on low-level modules. Both should depend on abstractions. B) Abstractions should not depend upon details. Details should depend upon abstractions.
- The Interface Segregation Principle (ISP) - Clients should not be forced to depend on methods they do not use.
- Principles of Least Knowledge (PLK) - Talk to your immediate friends.
- Principle of Loose Coupling - object that interact should be loosely coupled with well-defined intefaces.

7.1.2. Design Patterns

"Design patterns make it easier to resuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new sysntems. Desugb patterns help to choose design alternatives that make a system reusable and avoid alternatives that compromise resuability. "[36]

Creational Design Patterns abstract the instantiation process and provide indepedecne for object creation, composition and representation. Factory - "The Factory Design Pattern is probably the most used design pattern in modern programming languages like Java and C#. It comes in different variants and implementations. If you are searching for it, most likely, you'll find references about the GoF patterns: Factory Method and Abstract Factory.

- creates objects without exposing the instantiation logic to the client.
- refers to the newly created object through a common interface

"[16]

Abstract Factroy(for HO test sheets) - "Using this pattern a framework is defined, which produces objects that follow a general pattern and at runtime this factory is paired with any concrete factory to produce objects that follow the pattern of a certain country. In other words, the Abstract Factory is a super-factory which creates other factories (Factory of factories). Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes."[16]

ObjectPool (passes sheet object through streams) - "pattern offer a mechanism to reuse objects that are expensive to create. . "[16]

Behavior Patterns - bahaviour patters are concerned with algorithms and the assignmens of resposnibilities between objects. Beahvioral pattersndescribe not just patterns of objects or classes but also the patterns of communication between them, These patterns characterize complex control flow thats difficult to tfollow at run time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected"[36] Interpreter (translation from ts to js) - "The Interpreter is one of the Design Patterns published in the GoF which is not really used. Ussualy the Interpreter Pattern is described in terms of formal grammars, like it was described in the original form in the GoF but the area where this design pattern can be applied can be extended.

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the lan-

guage.

- Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design

"[16]
Strategy (multipiping/piping of streams) - "lets the algorithm vary independently from clients that use it."[16]
Observer (event emitor) - "The Observer Design Pattern can be used whenever a subject has to be observed by one or more observers."[16] Visitor (callbacks in JS) - "

- Represents an operation to be performed on the elements of an object structure.
- Visitor lets you define a new operation without changing the classes of the elements on which it operates.

"[16]

7.2. Functional Programming

Functional Programming languages are languages which supports functions as a first-class citizens. Which mean that languge provides an opotunity to store them in data strucutures, pass and return them from functions they are higher-order functions[38].

Declarative languages in contrast to imperative ones are characterized as having no implicit state. Functional languages are declarative languages whose underlaying computational model is the function.[38]

The most fundamental influence developing of functional languages was the work of Alnso Church on lambda calculus. "Church's lambda calculus was the first suitable treatment of the computational aspects of functions."[38]

Introduction of lambda functions to Java SE 8 as a new and important feature[**?**] indicates the growing need of imperative programming benefits in an Enterprise

Software development.

"Modeling with objects is powerful and intuitive, largely because this matches the perception of interacting with a world of which we are part. However, as we've seen repeatedly throughout this chapter, these models raise thorny problems of constraining the order of events and of synchronizing multiple processes. The possibility of avoiding these problems has stimulated the development of functional programming languages, which do not include any provision for assignment or mutable data. In such a language, all procedures implement well-defined mathematical functions of their arguments, whose behavior does not change. The functional approach is extremely attractive for dealing with concurrent systems." [28]

"John Backus, the inventor of Fortran, gave high visibility to functional programming when he was awarded the ACM Turing award in 1978. His acceptance speech (Backus 1978) strongly advocated the functional approach. A good overview of functional programming is given in Henderson 1980 and in Darlington, Henderson, and Turner 1982."[28]

7.3. Streams

"A stream is an abstract interface implemented by various objects in Node.js." [18] "Dominic Tarr (one of top contributors to the Node.js community [22]), defines streams as node's best and most misunderstood idea."[33]

Streams are the classic example of Pipe-and-filter architecture.

"In an event-based platform such as Node.js, the most efficient way to handle I/O is in real time, consuming the input as soon as it is available and sending the output as soon as it is produced by the application."[33]

- Spatial efficiency
- Time efficiency
- Composability

7.3.1. Anatomy of Streams

"Every stream in Node.js is an implementation of one of the four base abstract classes available in the stream core module:

- stream.Readable
- stream.Writable
- stream.Duplex
- stream.Transform

Each stream class is also an instance of EventEmitter. Streams, in fact, can produce several types of events, such as end, when a Readable stream has finished reading, or error, when something goes wrong.

One of the reasons why streams are so flexible is the fact that they can handle not only binary data, but practically, almost any JavaScript value; in fact they can support two operating modes:

- Binary mode: This mode is where data is streamed in the form of chunks, such as buffers or strings;
- Object mode: This mode is where the streaming data is treated as a sequence of discreet objects (allowing to use almost any JavaScript value).

Readable streams. A readable stream represents a source of data; in Node.js, it's implemented using the Readable abstract class that is available in the stream module.

Writable streams. A writ[e]able stream represents a data destination; in Node.js, it's implemented using the Writ[e]able abstract class, which is available in the stream module.

Duplex streams. A Duplex stream is a stream that is both Readable and Writ[e]able. It is useful when we want to describe an entity that is both a data source and a data destination, as for example, network sockets. Duplex streams inherit the methods of both stream.Readable and stream.Writable, so this is nothing new to us. This means that we can read() or write() data, or listen for both the readable and drain events.

Transform streams. The Transform streams are a special kind of Duplex stream that are specifically designed to handle data transformations. In a simple Duplex stream, there is no immediate relationship between the data read from the stream and the data written into it (at least, the stream is agnostic to such a relationship). On the other side, Transform streams apply some kind of transformation to each chunk of data that they receive from their Writable side and then make the transformed data available on their Readable side. From the outside, the interface of a Transform stream is exactly like that of a Duplex stream. However, when we want to build a new Duplex stream we have to provide the read() and write() methods while, for implementing a new Transform stream, we have to fill in another pair of methods: transform() and flush()."[33]

7.3.2. Piping patterns

"As in real-life plumbing, Node.js streams also can be piped together following different patterns; we can, in fact, merge the flow of two different streams into one, split the flow of one stream into two or more pipes, or redirect the flow based on a condition. In this section, we are going to explore the most important plumbing techniques that can be applied to Node.js streams."[33]

- Combining streams - 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.[33]
- Forking streams - piping single readable in to multiple writable streams.[33]
- Merging streams - piping multiple readable streams in to single writable stream.[33]
- Multiplexing and demultiplexing - forking and merging pattern which provides shared communication chanel for entities from different sreams.[33]

8. Contributions

For particular implementation of Test Sheets paradigm were developed conventions conventions described below.

8.1. Requirements analysis

This section describes and analyses requirements differences defined by Test Sheets concept and introduced by figo GmbH.

Test Sheets were originally designed for test definitions of OOP languages. And all available examples describes tests definitions for Java Classes. While figo GmbH case requires implementation of tests for nodeJS/casperJS, which are based on JavaScript - Object Oriented, imperative, Functional Oriented programming language with asynchronous information flow. Moreover figo GmbH requires input of test results to be recorded in to LogStash logs database. The execution requirements from figo GmbH are such that tests defined via Test Sheets should be automatically executed in a time manner (every 2 mins or so), while normal software testing is performed on demand. The figo's requirement for comparison is such that while defining a Test Sheet user should be able to select from two types of comparison: 1) Strict comparison - complete comparison of objects including both properties' structure and their values. 2) Not-Strict comparison - scheme comparison of objects.

(Probably should go to different section) Scraping scripts implement callback based approach for handling asynchronous data flow. This provides an opportunity to perform result comparison within the custom callback defined on a implementation stage. Standard convention for callback definitions limitates number of input parameters of a callback (error, data), while comparison requires compare data parameter with expected outputs from Test Sheet. The opportunity to resolve this issue lays in a JavaScript's support of functions as a class citizens, the function implementation of module for comparison and report: module

exports function which invoked with single parameters (expected_output) / (+ script name) and returns function which is used as a callback for scrapping script, this callback function performs comparison and writes its result to TestSheet or logstash depending on environment in which the program was executed

8.2. Conventions

Following conventions should be followed for Test Sheet passed verification.

General:

- Number of columns within one TS should not exceed 26 columns (from A to Z);
- Invocation delimiters must be allocated within single column the (aligned to the longest row);
- References to the columns with expected returns columns will take as value actual return value obtained from method execution;
- Files extensions should be .xlsx

Basic Test Sheets

- A1 cell(optional) - description of the test case;
- A2 cell - module under testing with an extension (.js);
- A3..n - name of the class/object under the test;
- B3..n - name of the method from representative class (same row) under the test;
- C2..n to Invocation Column - input parameters for representative method (same row) under the test;
- 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(s) after invocation line.

Parameterized and Higher-Order Test Sheets Lower order test sheets can belong to Basic of Non-Linear types of Test Sheets and respectively follow conventions, with next additional option:

- Input and/or output cells can contain parameters ?[B-Z]+ which represent the value of cells within the representative column of Higher-Order Test Sheet
- Rows 1 and 2 should follow conventions for Basic Test Sheet;
- Cells starting from second row inside of [B-Z] columns should contain values which will replace parameters inside of Parameterized Test Sheet.

8.3. Use Case

definition (possibly in tabular form)

- Test Sheets defined by users (clients or employees without development background).
- Tests themselves will be applied for identification of layout changes on a target page before any interaction will appear to avoid errors and minimize the time of scripts correction.

Execution stages:

- Automated transformation of Test Sheets into JavaScript tests;
- Scheduled task for running tests on web pages;
- Developer notification regarding failing test.

The program run by user

8.4. Architecture

This system implements Pipe-and-Filter Architecture. [35] [33]

"In a pipe-and-filter style architecture, the computations components are called filters and they act as transducers that take input, transform it according to one or mode algorithms, and then output the result to communications conduit. The input and outputs coduits are called pipes.
The filters must be intedependent components. [...] The clasic example of pipe-and-filter architectural style is the Unix shell[...]"[35].

8.5. Design and Implementation

Consists of two streams Reader and Writer both streams are in object mode.

The system's information workflow described with following explanatory Test Sheet:

	A	B	C	D	E	F
1	get Accounts					
2	BankAustria					
3	BankAustria	login	<credentials object>	<pin object>	|	{...}
4	BankAustria	getAccounts	<credentials object>		||	{...}

8.5.1. Reader

On a lower level Reader stream consists of two combined streams (streams combination pattern used);

First stream takes input as a directrory and returns list of absolute paths to all files within provided directory (including nested folders); Second stream accepts output of a first stream and for all .xlsx files obtains its schema invoking function from schema_maker library and as an output returns object with absolute path to the Test Sheet file with file content returned by reading with object returned by reading file (object pool pattern) with *xlsx* library (`https://www.npmjs.com/package/xlsx`) and scheme created by schema_maker library.
Schema structure for expample Test Sheet:

- pathToFile: 'absloute/path/to/test/sheet/file';
- testsheet: { < object returned by xlsx library >};
- schema:

description: 'get Accounts',

moduleUnderTest: 'BankAustria',

objectsUnderTest: ['A3', 'A4'],

methodsUnderTest: ['B3', 'B4'],

inputs: ['C3', 'C4', 'D3],

outputs: ['F3', 'F4'],

invocations: ['E3', 'E4'],

Correspondence to design principles:

- Closing - stream is closed over file extension, schema_maker - object structure returned by *xlsx* library;
- Code to Interface - stream obtains and returns values via standard stream interface, call to file system made via standard nodeJS File System stream interface;
- Do not Repeat Yourself - no code duplication;
- Single Responsibility Principle - can be changed only due to the change of input type;
- Open Close Principle - new pipes can be added in a single place;
- Liscov Substitution Principle - no inherited objects used;
- Interface Segregation Principle - no dependency on redundant methods;
- Dependency Inversion Principle - higher level module index.js does not depend on current library
- Least Knowledge Principle - communication to interfaces and invocation of used library;
- Loose Coupling Principle - standard interfaces;

8.5.2. Writer

9. Implementierung

Zusammen mit dem Betreuer werden use-cases entwickelt anhand deren die Software programmiert werden soll. Diese dienen auch als Bewertungsgrundlage.

Die allgemein empfohlene Verzeichnisstruktur eines Projektes sieht wie folgt aus:

- projektname
  - bin
  - doc
  - lib
  - src

Die zu erstellende Software soll im package de.uni_mannheim.informatik.swt.`projektname` unter `src` liegen.

Bei der Programmierung sollte durchgängig die englische Sprache verwendet werden. Hierzu zählen insbesondere Kommentare im Quellcode, Namen von Funktionen, Variable, Menüpunkte im Benutzerinterface, kurze Hilfestellungen und Ausgaben von Programmen.

Der Code sollte mit Hilfe von `lstinputlisting` formatiert und ausgegeben werden, wie in folgendem Beispiel:

```
package de.uni_mannheim.informatik.swt.projektname;

import javax.servlet.http.HttpServletRequest;

import de.unimannheim.wifo3.cobana.action.ActionForm;
import de.unimannheim.wifo3.cobana.action.ActionMapping;

public class GuestbookForm extends ActionForm {
    private String name = null;
```

```
10      private String message = null;

        public GuestbookForm() {
        }

15      public String getName() {
            return this.name;
        }
        public void setName(String name) {
            this.name = name;
20      }

        public String getMessage() {
            return this.message;
        }
25      public void setMessage(String message) {
            this.message = message;
        }

        public void reset(ActionMapping mapping, HttpServletRequest
            request) {
30          setName(null);
            setMessage(null);
        }

    }
```

Listing 9.1: GuestbookForm.java

Bibliography

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

[2] Analysis of generators and other async patterns in node.

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

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

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

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

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

[8] figo api reference. http://docs.figo.io/.

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

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

[11] Ieee sa - 829-2008 - ieee standard for software and system test documentation.

[12] Iintroduction to test driven development (tdd). http://www.agiledata.org/essays/tdd.html.

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

[14] Javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[15] 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.

[16] Object oriented design patterns. http://www.oodesign.com/.

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

[18] Stream: Node.js v5.4.1 documentation.

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

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

[21] Test sheets. http://swt.informatik.uni-mannheim.de/de/research/research-topics/test-sheets/parameterized-and-higher-order-test-sheets/.

[22] Top npm contributors by number of packages.

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

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

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

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

[27] 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-.

[28] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[29] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[30] Dave Astels. *Test Driven Development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003.

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

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

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

[34] Érika Cota. Embedded software testing: What kind of problem is this? In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1486–1486, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

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

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

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

[38] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, September 1989.

[39] P.C. Jorgensen. *Software Testing: A Craftsman's Approach, Fourth Edition*. An Auerbach book. Taylor & Francis, 2013.

[40] Michael R. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[41] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

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

[43] Robert W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley Publishing Company, USA, 9th edition, 2009.

[44] Mladen A Vouk et al. Software reliability engineering. 2000.

Appendix

A. First class of appendices

A.1. Some appendix

This is a sample appendix entry.

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, February 19, 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, February 19, 2016 Unterschrift