Using Test Sheets for Real Time Testing in the case of Figo GmbH

Denys Zaliskyi[1]

University of Mannheim, Germany
dzaliskyi@informatik.uni-mannheim.de

Abstract. This paper describes processes of design and implementation of the Test Sheets' concept and an integration of the product to business processes of figo GmbH for a real-time testing/validation of internet banking web pages with application of scraping scripts.
Implications of this research paper will be helpfull for students of computer scince disciplines who are interested in combined application of best practices for system design and development from both Object Oriented Programming (OOP) and Functional Programming (FP) approaches. In this paper they will find analysis of suggestions, principles and patterns for design and implementation of scalable and reusable software together with detailed description of the design and implementation process of the Test Sheets paradigm for asyncronous real-time testing.
Paper also defines conventions for Test Sheets and indicates test requirements for test implementation and execution.
The design process was made with respect to NodeJS best practices and integral design aspects, design principles and design patterns for OOP together with pipe-and-filter architectural type and application of piping strategies of a FP.
Implementation process described in this paper was performed with following of the Test Driven Development (TDD) rules which guarantees full test coverage together with Clean Code recomendations by Robert C Martin and coding style guide introduced by Airbnb.

Keywords: nodejs, design patterns, design principles, solid, streams, asynchronous, testing, test driven development, test sheets

1 Introduction

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 Testing

Introduction of Agile and Extreme developmet paradigms with increased requirenments for control over the existing code, its reuse and maintanace lifted requirements for tests and thei quality to the new level and shifted responsibility regarding testing process from software engineers to people without development background. This led to new requirenments 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 wirte 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 exibility, maintainability, and reusability of the production code. "[25]

"The ideas and techniques of software testing have become essential knowledge for all software delopers"[14]

Testing is responsible for at least 30% of the cost in a software project.[17]

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. [16]

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) [16]

2.1 Test Driven Development

Test Driven Development is a software development paradigm which combines test-first development and refactroing. This means that programmers frist define test and only after write code to fullfil test requirenments, both code and test maintained by developers. This reduces amount of redundant code and improves developers' confidence during refactoring process. Refactroing in TDD requires developer before introducing new feature first ask question the existing design fits best for implementation of new functionality.[2] [15]

"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.[25]

Pros. As a result you will always be improving the quality of your design, thereby making it easier to work with in the future.[2] Excelent fault isolation. Support by big variety of test frameworks [24] Tests can replace code documentation at certain level of abstraction [25]

Cons. Very hard to perform TDD without testing framework. [24] Also [24] states among the limitations of TDD is its bottom-up nature which provides little oportunity for elegant design. In contrast [2] 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 [24] claims that some of the faults can be tracked only by data flow testing.

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

Fast. Tests should be fast. They should run quickly. When tests run slow, you wont want to run them frequently. If you dont run them frequently, you wont find problems early enough to fix them easily. You wont feel as free to clean up the code. Eventually the code will begin to rot.[25]

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.[25]

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 arent repeatable in any environment, then youll always have an excuse for why they fail. Youll also nd yourself unable to run the tests when the environment isnt available.[25]

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 les to see whether the tests pass. If the tests arent self-validating, then failure can become subjective and running the tests can require a long manual evaluation.[25]

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 nd 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.[25]

3 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."[8]

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."[9]

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"[10]

3.1 Scenario Testing

"Scenario testing is defined as a 'set of realistic user activities that are used for evaluating the product'. It is also defined as the testing involving customer scenarios."[19] By definition of Test Sheets user defines Story Line[19] and Life Cycle[19] and free to create Battle Ground[19] by setting expected error outputs.

3.2 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."[22] (MORE SHOULD BE ADDED)

4 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-changerthe 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. "[18]

"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."[5]

4.1 Event handling

NodeJS asynchronous nature provided by event handler which is an implementation of reactor pattern. Here is the description of process lifecycle[18]:

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.

4.2 Non blocking I/O

A set of bindings responsible for wrapping and exposing libuv and other low-level functionality to JavaScript.[18]

Non blocking I/O in NodeJS is provided by libuv[1][18]. 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 its also used by Luvit, Julia, pyuv, and others."[12]
libuv properties[3]:

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

4.3 "Node core

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

4.4 V8/Chakra

the JavaScript engine originally developed by Google for the Chrome browser/ Microsoft for IE 9 browser"[18]

5 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. "[27]

5.1 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.[26]

Symptoms of not agile design [26]:

Rigidity (The design is difficult to change) - Rigidity is the tendency for software to be difficult to change, even in simple ways. A design is rigid if a single change causes a cascade of subsequent changes in dependent modules. The more modules that must be changed, the more rigid the design. Most developers have faced this situation in one way or another. They are asked to make what appears to be a simple change. They look the change over and make a reasonable estimate of the work required. But later, as they work through the change, they find that there are unanticipated repercussions to the change. The developers find themselves chasing the change through huge portions of the code, modifying far more modules than they had first estimated, and discovering thread after thread of other changes that they must remember to make. In the end, the changes take far longer than the initial estimate. When asked why their estimate was so poor, they repeat the traditional software developers lament: "It was a lot more complicated than I thought!"

Fragility (The design is easy to break) - Fragility is the tendency of a program to break in many places when a single change is made. Often, the new problems are in areas that have no conceptual relationship with the area that was changed. Fixing those problems leads to even more problems, and the development team begins to resemble a dog chasing its tail. As the fragility of a module increases, the likelihood that a change will introduce unexpected problems approaches certainty. This seems absurd, but such modules are not at all uncommon. These are the modules that are continually in need of repair, the ones that are never off the bug list. These modules are the ones that the developers know need to be redesigned, but nobody wants to face the spectre of redesigning them. These modules are the ones that get worse the more you fix them.

Immobility (The design is difficult to reuse) - A design is immobile when it contains parts that could be useful in other systems, but the effort and risk involved with separating those parts from the original system are too great. This is an unfortunate but very common occurrence.

Viscosity (It is difficult to do the right thing) - Viscosity comes in two forms: viscosity of the software and viscosity of the environment. When faced with a change, developers usually find more than one way to make that change. Some of the ways preserve the design; others do not (i.e., they are hacks). When the design-preserving methods are more difficult to use than the hacks, the viscosity of the design is high. It is easy to do the wrong thing but difficult to do the right thing. We want to design our software such that the changes that preserve the design are easy to make. Viscosity of environment comes about when the development environment is slow and inefficient. For example, if compile times are very long, developers will be tempted to make changes that don't force large recompiles, even though those changes don't preserve the design. If the source code control system requires hours to check in just a few files, developers will be tempted to make changes that require as few check-ins as possible, regardless of whether the design is preserved. In both cases, a viscous project is one in which

the design of the software is difficult to preserve. We want to create systems and project environments that make it easy to preserve and improve the design.

Needless complexity (Overdesign) - A design smells of needless complexity when it contains elements that aren't currently useful. This frequently happens when developers anticipate changes to the requirements and put facilities in the software to deal with those potential changes. At first, this may seem like a good thing to do. After all, preparing for future changes should keep our code flexible and prevent nightmarish changes later. Unfortunately, the effect is often just the opposite. By preparing for many contingencies, the design becomes littered with constructs that are never used. Some of those preparations may pay off, but many more do not. Meanwhile, the design carries the weight of these unused design elements. This makes the software complex and difficult to understand.

Needless repetition (Mouse abuse) - Cut and paste may be useful text-editing operations, but they can be disastrous code-editing operations. All too often, software systems are built on dozens or hundreds of repeated code elements. It happens like this: Ralph needs to write some code that fravles the arvadent. He looks around in other parts of the code where he suspects other arvadent fravling has occurred and finds a suitable stretch of code. He cuts and pastes that code into his module and makes the suitable modifications. Unbeknownst to Ralph, the code he scraped up with his mouse was put there by Todd, who scraped it out of a module written by Lilly. Lilly was the first to fravle an arvadent, but she realized that fravling an arvadent was very similar to fravling a garnatosh. She found some code somewhere that fravled a garnatosh, cut and paste it into her module, and modified it as necessary. When the same code appears over and over again, in slightly different forms, the developers are missing an abstraction. Finding all the repetition and eliminating it with an appropriate abstraction may not be high on their priority list, but it would go a long way toward making the system easier to understand and maintain. When there is redundant code in the system, the job of changing the system can become arduous. Bugs found in such a repeating unit have to be fixed in every repetition. However, since each repetition is slightly different from every other, the fix is not always the same.

Opacity (Disorganized expression) - Opacity is the tendency of a module to be difficult to understand. Code can be written in a clear and expressive manner, or it can be written in an opaque and convoluted manner. Code that evolves over time tends to become more and more opaque with age. A constant effort to keep the code clear and expressive is required in order to keep opacity to a minimum. When developers first write a module, the code may seem clear to them. After all, they have immersed themselves in it and understand it at an intimate level. Later, after the intimacy has worn off, they may return to that module and wonder how they could have written anything so awful. To prevent this, developers need to put themselves in their readers' shoes and make a concerted effort to refactor their code so that their readers can understand it. They also need to have their code reviewed by others.

Foundamental Object Oriented design principles:

Closing - Encapsulate things in your design that are likely to change. This means to protect your classes from unnecessary change by separating the features and methods of a class that relatively constant throughout the program from that will change. By separating the two types of features, we isolate the parts that will change a lot into a separate class (or classes) that we can depend on changing, and we increase our flexibility and ease of change. [20]

Code to an Interface rather then to an implementation. [20]

Do not repeat yourself (DRY) - Avoid duplicate code. Whenever you find common behaviour in two or more places, look to abstract tht behaviour into a class and then resue that behaviour in the common concrete classes. Satisfy one requirement in one place in your code[20]

The Single-Responsibility Principle (SRP) - A class should have only one reason to change. In the context of the SRP, we define a responsibility to be a reason for change. If you can think of more than one motive for changing a class, that class has more than one responsibility. This is sometimes difficult to see. We are accustomed to thinking of responsibility in groups. The Single-Responsibility Principle is one of the simplest of the principles but one of the most difficult to get right. Conjoining responsibilities is something that we do naturally. Finding and separating those responsibilities is much of what software design is really about. Indeed, the rest of the principles we discuss come back to this issue in one way or another.[26][20]

The Open/Closed Principle (OCP) - Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. When a single change to a program results in a cascade of changes to dependent modules, the design smells of rigidity. OCP advises us to refactor the system so that further changes of that kind will not cause more modifications. If OCP is applied well, further changes of that kind are achieved by adding new code, not by changing old code that already works. This may seem like motherhood and apple piethe golden, unachievable idealbut in fact, there are some relatively simple and effective strategies for approaching that ideal. Modules that conform to OCP have two primary attributes. They are open for extension. This means that the behavior of the module can be extended. As the requirements of the application change, we can extend the module with new behaviors that satisfy those changes. In other words, we are able to change what the module does. 1. They are closed for modification. Extending the behavior of a module does not result in changes to the source, or binary, code of the module. The binary executable version of the modulewhether in a linkable library, a DLL, or a .EXE fileremains untouched. 2. It would seem that these two attributes are at odds. The normal way to extend the behavior of a module is to make changes to the source code

of that module. A module that cannot be changed is normally thought to have a fixed behavior.

In many ways, the Open/Closed Principle is at the heart of object-oriented design. Conformance to this principle is what yields the greatest benefits claimed for object-oriented technology: flexibility, reusability, and maintainability. Yet conformance to this principle is not achieved simply by using an object-oriented programming language. Nor is it a good idea to apply rampant abstraction to every part of the application. Rather, it requires a dedication on the part of the developers to apply abstraction only to those parts of the program that exhibit frequent change. Resisting premature abstraction is as important as abstraction itself.

The Liskov Substitution Principle - Subtypes must be substitutable for their base types. The Open/Closed Principle is at the heart of many of the claims made for object-oriented design. When this principle is in effect, applications are more maintainable, reusable, and robust. The Liskov Substitution Principle is one of the prime enablers of OCP. The substitutability of subtypes allows a module, expressed in terms of a base type, to be extensible without modification. That substitutability must be something that developers can depend on implicitly. Thus, the contract of the base type has to be well and prominently understood, if not explicitly enforced, by the code. The term IS-A is too broad to act as a definition of a subtype. The true definition of a subtype is substitutable, where substitutability is defined by either an explicit or implicit contract.[26][20]

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.

Consider the implications of high-level modules that depend on low-level modules. It is the high-level modules that contain the important policy decisions and business models of an application. These modules contain the identity of the application. Yet when these modules depend on the lower-level modules, changes to the lower-level modules can have direct effects on the higher-level modules and can force them to change in turn. This predicament is absurd! It is the high-level, policy-setting modules that ought to be influencing the low-level detailed modules. The modules that contain the high-level business rules should take precedence over, and be independent of, the modules that contain the implementation details. Highlevel modules simply should not depend on low-level modules in any way. Moreover, it is high-level, policy-setting modules that we want to be able to reuse. We are already quite good at reusing low-level modules in the form of subroutine libraries. When high-level modules depend on low-level modules, it becomes very difficult to reuse those high-level modules in different contexts. However, when the high-level modules are independent of the low-level modules, the highlevel modules can be reused quite simply. This principle is at the very heart of framework design.

Traditional procedural programming creates a dependency structure in which policy depends on detail. This is unfortunate, since the policies are then vulner-

able to changes in the details. Objectoriented programming inverts that dependency structure such that both details and policies depend on abstraction, and service interfaces are often owned by their clients. Indeed, this inversion of dependencies is the hallmark of good object-oriented design. It doesn't matter what language a program is written in. If its dependencies are inverted, it has an OO design. If its dependencies are not inverted, it has a procedural design. The principle of dependency inversion is the fundamental low-level mechanism behind many of the benefits claimed for object-oriented technology. Its proper application is necessary for the creation of reusable frameworks. It is also critically important for the construction of code that is resilient to change. Since abstractions and details are isolated from each other, the code is much easier to maintain.[26][20]

The Interface Segregation Principle (ISP) - Clients should not be forced to depend on methods they do not use. When clients are forced to depend on methods they don't use, those clients are subject to changes to those methods. This results in an inadvertent coupling between all the clients. Said another way, when a client depends on a class that contains methods that the client does not use but that other clients do use, that client will be affected by the changes that those other clients force on the class. We would like to avoid such couplings where possible, and so we want to separate the interfaces.

This principle deals with the disadvantages of "fat" interfaces. Classes whose interfaces are not cohesive have "fat" interfaces. In other words, the interfaces of the class can be broken up into groups of methods. Each group serves a different set of clients. Thus, some clients use one group of methods, and other clients use the other groups. ISP acknowledges that there are objects that require noncohesive interfaces; however, it suggests that clients should not know about them as a single class. Instead, clients should know about abstract base classes that have cohesive interfaces.

Fat classes cause bizarre and harmful couplings between their clients. When one client forces a change on the fat class, all the other clients are affected. Thus, clients should have to depend only on methods that they call. This can be achieved by breaking the interface of the fat class into many client-specific interfaces. Each client-specific interface declares only those functions that its particular client or client group invoke. The fat class can then inherit all the client-specific interfaces and implement them. This breaks the dependence of the clients on methods that they don't invoke and allows the clients to be independent of one another.[26][20]

Principles of Least Knowledge (PLK) - Talk to your immediate friends. The complement to strong cohesion in an application is loose coupling. Thats what the Principle of Least Knowledge is all about. It says that classes should collaborate indirectly with as few other classes as possible.This leads us to a corollary to the PLK keep dependencies to a minimum. This is the crux of loose coupling. By interacting with only a few other classes, you make your class more flexible and less likely to contain errors. [20]

Principle of Loose Coupling - object that interact should be loosely coupled with well-defined intefaces.[20]

5.2 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. "[21]

Creational Design Patterns abstract the instantiation process and provide independecne 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

"[6]

Abstract Factroy(for HO test sheets) - "Modularization is a big issue in today's programming. Programmers all over the world are trying to avoid the idea of adding code to existing classes in order to make them support encapsulating more general information. Take the case of a information manager which manages phone number. Phone numbers have a particular rule on which they get generated depending on areas and countries. If at some point the application should be changed in order to support adding numbers form a new country, the code of the application would have to be changed and it would become more and more complicated.

In order to prevent it, the Abstract Factory design pattern is used. 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."[6]

ObjectPool (passes sheet object through streams) - "erformance can be sometimes the key issue during the software development and the object creation(class instantiation) is a costly step. While the Prototype pattern helps in improving the performance by cloning the objects, the Object Pool pattern offer a mechanism to reuse objects that are expensive to create.

Clients of an object pull "feel" like they are owners of a service although the service is shared among many other clients.

-reuse and share objects that are expensive to create. "[6]

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"[21] 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 language.
- Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design

"[6]
Strategy (multipiping/piping of streams) - "There are common situations when classes differ only in their behavior. For this cases is a good idea to isolate the algorithms in separate classes in order to have the ability to select different algorithms at runtime. Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."[6]
Observer (event emitor) - "We can not talk about Object Oriented Programming without considering the state of the objects. After all object oriented programming is about objects and their interaction. The cases when certain objects need to be informed about the changes occured in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Observer Design Pattern can be used whenever a subject has to be observed by one or more observers.

Let's assume we have a stock system which provides data for several types of client. We want to have a client implemented as a web based application but in near future we need to add clients for mobile devices, Palm or Pocket PC, or to have a system to notify the users with sms alerts. Now it's simple to see what we need from the observer pattern: we need to separate the subject(stocks server) from it's observers(client applications) in such a way that adding new observer will be transparent for the server. Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."[6] Visitor (callbacks in JS) - "Collections are data types widely used in object oriented programming. Often collections contain objects of different types and in those cases some operations have to be performed on all the collection elements without knowing the type. A possible approach to apply a specific operation on objects of different types in a collection would be the use if blocks in conjunction with 'instanceof' for each element. This approach is not a nice one, not flexible and not object oriented at all. At this point we

should think to the Open Close principle and we should remember from there that we can replace if blocks with an abstract class and each concrete class will implement its own operation.

- 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.

"[6]

6 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[23].

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.[23]

The most fundamental influence developing of functional languages was the work of Alnso Church on lambda calculus. "Churchs lambda calculus was the first suitable treatment of the computational aspects of functions."[23]

Introduction of lambda functions to Java SE 8 as a new and important feature[4] 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." [13]

"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."[13]

6.1 Streams

"A stream is an abstract interface implemented by various objects in Node.js." [7]
"Dominic Tarr (one of top contributors to the Node.js community [11]), defines streams as node's best and most misunderstood idea."[18]

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."[18]

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." [18]

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."[18]

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."[18]

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()."[18]

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."[18]

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.[18]

Forking streams - piping single readable in to multiple writable streams.[18]

Merging streams - piping multiple readable streams in to single writable stream.[18]

Multiplexing and demultiplexing - forking and merging pattern which provides shared communication chanel for entities from different sreams.[18] [18]

7 Literature Review

8 Contributions

8.1 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)
- file extension .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.

8.2 Architecture

This system implements Pipe-and-filer Archtecture with some external complexity included in to piping mechanisms with application of pipeing patterns. [20] [18]

"In a pipe-and-filter style acrchitecture, the computtions components are called filters and they act as transducers that take input, transform it accroding 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. That is one of the bauties of a pipe -and-filter architecture. [...] The clasic example of pipe-and-filter architectural style is the Unix shell[...]"[20].

This is a Master Thesis project for implementation of a software for the generation of a real-time web-page tests from a provided Test Sheets for figo GmbH.

Use Case difinition (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.

8.3 Design

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

Reader accepts directory with Test Sheets as an input paramter and retruns a Schema object with a following struncture for all directrory entries to the standard stream interface.

Schema structure:

- description: 'A1',
- moduleUnderTest: 'A2',
- objectsUnderTest: ['A3', 'A4', 'A5'],
- methodsUnderTest: ['B3', 'B4', 'B5'],
- inputs: ['C3', 'C4', 'C5', 'D3', 'D4', 'D5'],
- outputs: ['F3', 'F4', 'F5', 'G3', 'G4', 'G5']

Reader structure:
On a lower level Reader stream consists fo two combined streams (streams combination pattern used); First stream takes input as a directrory and returns list of all files within provided directory (including nested folders); Second stream accepts output of a first stream and for all .xlsx files obtains its schema invoceing function from schema_maker library and as an output returns object with a following structure

!!!write about principles applied to each module, design patterns used in each module

Reader - stream object;
scheme_generator

Reader stream output structure:

- address: 'path to .xlsx file',
- scheme: 'schema object',
- sheet: '.xlsx file content'

9 Limitations and Future Works

10 Related Work

11 Conclusions

References

1. About node.js, https://nodejs.org/en/about/
2. Iintroduction to test driven development (tdd), http://www.agiledata.org/essays/tdd.html
3. An introduction to libuv: Basics of libuv, http://nikhilm.github.io/uvbook/basics.html
4. Java se 8 lambda quick start, https://apexapps.oracle.com/pls/apex/f?p=44785:24:100775742404222::NO:24:P24_C
5. Javascript, https://developer.mozilla.org/en-US/docs/Web/JavaScript
6. Object oriented design patterns, http://www.oodesign.com/
7. Stream: Node.js v5.4.1 documentation, https://nodejs.org/docs/latest/api/stream.html
8. Test sheets, http://swt.informatik.uni-mannheim.de/de/research/research-topics/test-sheets/
9. Test sheets, http://swt.informatik.uni-mannheim.de/de/research/research-topics/test-sheets/basic-test-sheets/

10. Test sheets, http://swt.informatik.uni-mannheim.de/de/research/research-topics/test-sheets/parameterized-and-higher-order-test-sheets/
11. Top npm contributors by number of packages, https://gist.github.com/michalbe/71f6f9e5938eeb781dc4
12. Welcome to the libuv api documentation, http://docs.libuv.org/en/v1.x/#features
13. Abelson, H., Sussman, G.J.: Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, USA, 2nd edn. (1996)
14. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, New York, NY, USA, 1 edn. (2008)
15. Astels, D.: Test Driven Development: A Practical Guide. Prentice Hall Professional Technical Reference (2003)
16. Atkinson, C.: L1-introduction (2015)
17. Atkinson, C.: L2-testingintroduction (2015)
18. Casciaro, M.: NodeJS Design Patterns. Packt Publishing (2014)
19. Desikan, S., Ramesh, G.: Software Testing: Principles and Practice. Pearson Education Canada (2006), https://books.google.de/books?id=Yt2yRW6du9wC
20. Dooley, J.: Software Development and Professional Practice. Apress, Berkely, CA, USA, 1st edn. (2011)
21. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
22. Glass, R.L.: Real-time: The “lost world” of software debugging and testing. Commun. ACM 23(5), 264–271 (May 1980), http://doi.acm.org/10.1145/358855.358857
23. Hudak, P.: Conception, evolution, and application of functional programming languages. ACM Comput. Surv. 21(3), 359–411 (Sep 1989), http://doi.acm.org/10.1145/72551.72554
24. Jorgensen, P.: Software Testing: A Craftsmans Approach, Fourth Edition. An Auerbach book, Taylor & Francis (2013), https://books.google.de/books?id=6WlmAQAAQBAJ
25. Martin, R.C.: Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edn. (2008)
26. Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River, NJ, USA (2003)
27. Sebesta, R.W.: Concepts of Programming Languages. Addison-Wesley Publishing Company, USA, 9th edn. (2009)