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

Denys Zaliskyi[1]

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

Abstract. This paper describes processes of design and development of realisation of the concept of Test Sheets and integration in to business process of Figo GmbH for a Real Time testing.

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

1 Introduction

I give a small background why it is important what we are doing.

State what the problem is.

Here we write what we are doing in our paper.

The paper is structured as follows: In the next section (Section 2) we will explain the foundations of bla bla bla. Section 3 presents our contribution. [...] The work is closing with related work (Section 5) and conclusions (Section 6).

2 Foundations

2.1 Testing

2.2 Test Sheets

–!– DO I NEED TO EXPLAIN CONCEPTS ON EXAPMLES?! –!–
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.[2]

Basic. 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.

A Result Sheet is shown as well. It differs from the input in that it has a new name and that it references the original sheet's name. Also the cells with expected values have been colored. Green for cells in which the expected value matched the value returned after executing the test, red otherwise. In case of a value mismatched both the expected value and the actual value are shown in the cell.[...][3]

Non-Linear Similar to the invocation line (double line separating input parameters and expected return values) there is a double line below the last method invocation row. Below this line the behavior specification starts. Each row of the behavior specification represents a state of a state machine. The state machine starts in the first state that is being specified right below the double line and stops execution once a state is reached without any valid transitions.

Each column in a row specifies a transition. A transition consists of a guard, executed invocations and a subsequent state. The starting state has only one transition with no guard, the intermediate state has [...] transitions each with a guard and the terminating state does not have any transitions at all.

[...]**??**

High Order 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

2.3 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. It is loved not just for its technical capabilities, but also for the change of paradigm that it introduced 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. This aspect only enables scenarios such as single-language application stacks and sharing of code between the server and the client. Node.js itself is contributing to the rise and evolution of the JavaScript language. People realize that using JavaScript on the server is not as bad as it is in the browser, and they will soon start to love it for its pragmatism and for its hybrid nature, half way between object-oriented and functional programming. 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. Mutexes are replaced by queues, threads by callbacks and events, and synchronization by causality. 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. [4]

2.4 Functional Programming

2.5 Streams[4]

Streams are one of the most important components and patterns of Node.js. There is a motto in the community that says Stream all the things! and this alone should be enough to describe the role of streams in Node.js. Dominic Tarr, a top contributor to the Node.js community, defines streams as node's best and most misunderstood idea. There are different reasons that make Node.js streams so attractive; again, it's not just related to technical properties, such as

performance or efficiency, but it's something more about their elegance and the way they fit perfectly into the Node.js philosophy.

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.

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 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.

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. Let's demonstrate this by building application that we mentioned earlier, starting from the server side.

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.

With very little effort (just a few lines of code), we added an encryption layer to our application; we simply had to reuse an already available transform stream by including it in the pipeline that we already had. In a similar way, we can add and combine other streams, as if we were playing with Lego bricks. Clearly, the main advantage of this approach is reusability, but as we can see from the code we presented so far, streams also enable cleaner and more modular code. 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.

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.

Writ[e]able 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 Writ[e]able 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().

Async control flow with streams ?

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.

2.6 Object Oriented Programming

2.7 Design Patterns

2.8 Design Principles (S.O.L.I.D.)

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.

Symptoms of not agile design [6]:

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.

S.O.L.I.D:

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.

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.

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 vulnerable 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.

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.

2.9 Clean Code [5]

TDD By now everyone knows that TDD asks us to write unit tests first, before we write production code. But that rule is just the tip of the iceberg. 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.

Requirements for testing (F. I. R. S. T.) Clean tests follow ve other rules that form the above acronym:

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 nd problems early enough to x them easily. You wont feel as free to clean up the code. Eventually the code will begin to rot.

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 rst one to fail causes a cascade of downstream failures, making diagnosis difcult and hiding downstream defects.

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.

Self-Validating. The tests should have a boolean output. Either they pass or fail. You should not have to read through a log le 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.

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.

We have barely scratched the surface of this topic. Indeed, I think an entire book could be written about clean tests. 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. So keep your tests constantly clean. Work to make them expressive and succinct. Invent testing APIs that act as domain-specic language that helps you write the tests. If you let the tests rot, then your code will rot too. Keep your tests clean.

3 Contributions

aaaa aaaa aaa aaa aaaaa aaaaaaaaaaa aaaaaaaa aaaaaaa In Line **??** of Algorithm **??**.

4 Limitations and Future Works

5 Related Work

6 Conclusions

References

1. About node.js, https://nodejs.org/en/about/
2. Test sheets, http://swt.informatik.uni-mannheim.de/de/research/research-topics/test-sheets/
3. Test sheets, http://swt.informatik.uni-mannheim.de/de/research/research-topics/test-sheets/basic-test-sheets/
4. Casciaro, M.: NodeJS Design Patterns. Packt Publishing (2014)
5. Martin, R.C.: Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edn. (2008)
6. Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River, NJ, USA (2003)