Towards Continuous Delivery in System Integration Projects

Introducing a Strategy to Achieve Continuous Delivery and Test Automation with FitNesse

Sandra Drenthen

06-02-2014

UNIVERSITY OF TWENTE. everett.

Master's Thesis

Towards Continuous Delivery in System Integration Projects

Introducing a Strategy to Achieve Continuous Delivery and Test Automation with FitNesse

Author

Name	S. Drenthen BSc
Study	Master Student Computer Science, Software Engineering Department of Computer Science Software Engineering group University of Twente, Enschede, The Netherlands
Student Number	s0146110
E-mail	s.drenthen@alumnus.utwente.nl

Graduation Committee

First Supervisor	Dr. M.I.A. Stoelinga Associate Professor Department of Computer Science Formal Methods and Tools group University of Twente, Enschede, The Netherlands
Second Supervisor	C.M. Bockisch Assistant Professor Department of Computer Science Formal Methods and Tools group University of Twente, Enschede, The Netherlands
Everett Supervisor	Drs. René Mulder Identity and Access Management Architect Everett NL BV, Nieuwegein, The Netherlands

Preface

This master thesis is the result of the final research project for the Computer Science master at the University of Twente. This project was performed externally at Everett. During the research project, I have learned a lot about continuous delivery, test automation, research methodologies and the process of conducting research. Furthermore, since the project was performed externally, I have experienced Everett's projects which enabled me to experience the practical field of computer science as well as the theoretical field.

First of all, I would also like to thank my supervisors Mariëlle, Pascal (during the first months), Christoph (during the last months), and René. I had the privilege to meet them all together every month, and my Everett supervisor René even every week. During these meetings, they provided feedback on my work, asked critical questions, guided me in the right direction and René also helped me with the practice. I believe their commitment in the guidance really did improve my work.

Furthermore, I would like to thank Everett for making this research project possible and the (anonymous) client of Everett for enabling me to apply the designed strategy at their project. I would like to thank the colleagues of both companies for making me feel welcome and part of the team, for all the fun we had and for their interesting insights on the project. Special thanks to the colleagues who invested their time to apply and evaluate my strategy in the case study and to my carpool-buddy for the fun trips, driving me almost every day to the case study and back.

Finally, I would like to thank my friends and family who supported me during the whole process, especially when I wasn't able to see them very often. They were always there for me: they helped me moving, they coped with my busy agenda, they cheered me up when needed, they gave me give good advice and they provided a lot of fun in the evenings and weekends. Special thanks for Pim for reviewing my thesis during his busy schedule, giving useful insights and improvements.

I hope that you like reading this thesis and that Everett and others can benefit from the results.

Sandra Drenthen

Abstract

This thesis presents a strategy to introduce continuous delivery and test automation with FitNesse in system integration projects. It was designed for Everett's identity solutions projects, but is expected to be mostly applicable for other system integration projects as well.

The strategy consists of tools, tutorials, approaches and guidelines:

- NetBeans, Git, Ant, JUnit, FitNesse and Jenkins were chosen as tools, used to achieve continuous delivery and test automation.
- Tutorials were written for FitNesse, Jenkins and a solution to automatically push and pull to Git.
- Several approaches and guidelines were chosen and tailored to Everett's projects and showed how to incorporate test driven development, how to decide which tests should be automated, how FitNesse and Jenkins needs to be configured and used, how to cope with test data changes and how to communicate with the system under test.

The strategy was constructed by first creating a high level strategy, which was then applied, evaluated, supplemented and improved during a case study at a client of Everett. During the case study, ten automated tests were made by the team-members of the project and the continuous delivery cycle was set up.

Evaluations were held with team-members of the case study project, which showed that the strategy did need some time investments in creating the strategy and tests, but can deliver an added value as it is expected to reduce faults, increase the quality, enhance trust from the client and possibly in the long run, also save time. However, in order to show this with hard facts and significance, the strategy should be applied once again where the strategy is complete in the beginning and more tests are written during the whole project.

keywords: continuous delivery, test automation, case study, FitNesse

Contents

1 Introduction

1.1 Problem Statement

Everett's test strategy for their *system integration projects*, called *identity solutions* was ad-hoc and based on much manual work, resulting in high costs for testing and sometimes finding bugs in a late stadium (e.g., at the last sprints of the project). Everett wanted to reduce both the costs of testing and the amount of delivered faults by introducing *test automation* in their projects with the use of *FitNesse*. Furthermore, Everett wanted to automate and improve the process of software delivery further by introducing the *continuous delivery* pattern in their projects as well.

In order to set up the continuous delivery framework, several foundations are needed: good configuration management, automated build and deploy scripts, automated tests and a continuous delivery framework to manage the steps in the deployment pattern. In Everett's case, the introduction of test automation and a continuous delivery framework were the last two steps that needed to be taken in order to introduce continuous delivery in their projects.

Everett was founded in 1999 and has nearly 80 employees throughout the Netherlands, Italy and the United Kingdom [1]. Everett's mission is to help organizations around the world to be successful with identity solutions through consulting, system integration, and support services.

System integration projects are projects where different computing systems and software applications are linked together, physically or functionally, to act as a coordinated whole [2].

Identity solutions [3] is a name that Everett gives to their defined set of several solution areas[3]. Identity solutions projects are system integration projects which revolve around the scalable and timely management of users and their access to information and applications. These projects characterize themselves as consisting of highly configured and customized third party software and being highly integrated, data-driven and short term (10-15 weeks). Identity solutions projects are performed at different clients at several sectors and may be implemented with various techniques and third party products. Due to these characterizations, the projects are hard to test. The aspect that the projects are short term and different third party software is used for different projects gives less time and less reuse in the target of achieving a return on investment from test automation. See section 2.2 for more information on identity solutions.

1.2 Goal

The goal of this thesis is to introduce continuous delivery and, as an important part of continuous delivery, to introduce test automation with FitNesse in order to lower the costs on testing, reducing the amount of faults in projects and enhancing the process software delivery.

The designed strategy for test automation and continuous delivery defines the test and development process and consists of tool-selections, tutorials, approaches and guidelines. The strategy was documented in a wiki.

The requirements and evaluation criteria for the designed strategy were defined as follows:

- Efficacy - It must tackle the problem in the problem scope (i.e., enable parts of continuous delivery)
- Flexibility - It must be useable in different projects with different circumstances.
- Implementation time - It must be easy and fast to install, learn and use.
- Cost-effectiveness - It must have an early return on investment.
- Transferability - It must enable the client to keep using and maintaining the tool.

1.3 Research Questions

The main research question of this research project has been:

- *How can continuous delivery and test automation with FitNesse be introduced in system integration projects?*

This question was divided into the following underlining questions, which needed to be answered in order to provide an answer to the main research question:

1. *What is a good[1] strategy to introduce continuous delivery and test automation with FitNesse?*
    a. *Which guidelines and tools are used in this strategy?*
    b. *How will these guidelines and tools be tailored to system integration projects and to each other?*
2. *What is are the costs and benefits of applying the strategy?*
    a. *Which effects has the application of the strategy on a project?*
    b. *Is the strategy an improvement compared to the test- and development strategy used in earlier projects?*
    c. *Where is the break-even point to recoup the effort of applying this strategy?*
        i. *How does this differ in several factors of the projects (e.g. different test types, different features, different projects and different software)?*
    d. *To which extent is the strategy applicable for other system integration projects?*

1.4 Research Method

The first research question (i.e., what is a good strategy) has been answered by creating a high level strategy for test automation and continuous delivery based on literature by selecting promising combinations of methods, guidelines and tools and creating tutorials when needed. This strategy is described in chapter 5.

[1] The strategy is considered good when it satisfies the requirements mentioned in section 1.2

When the high level strategy was created, it was applied, evaluated, supplemented and improved during a case study which is discussed below.

The second research question (i.e., what are the costs and benefits) has been answered by evaluating the strategy in a case study. The case study approach allows to investigate the validity of the strategy in the real-life context of a identity solutions project in a qualitative manner. The case study followed an iterative research pattern, which has intermediate evaluations and strategy changes in order to design the strategy in steps and improve it along the way. The case study has been performed in five iterations, each during two weeks. Data was gathered from documents, observations, surveys and personal interviews. The validity of the strategy is determined by the outcome of the evaluations and personal interviews.

A more detailed research method description is given in chapter 3.

1.5 Scope

Three scopes have been defined in order to have some control on the size and the time needed to carry out the research project. First of all, his research used FitNesse as test automation tool, which was selected in earlier research as a promising tool to be used with Everett's projects. Furthermore, the strategy has been created for and applied on Everett's Identity & Access Governance projects, and even more specific, Identity & Access Governance projects using SailPoint IdentityIQ software. However this was a specific project and software choice, the goal was to be able to use (most of) the strategy for other projects and software as well.

1.5.1 Test Tool: FitNesse

In earlier research[4], several test automation tools have been compared and judged on the degree of how well they meet the requirements mentioned in section 1.2. The investigated tools were web browser automation tools like *Selenium* [5] and *Watir* [6] and test automation tools *FitNesse* [7], *GreenPepper* [8], *Cucumber* [9] and *Root Framework* [10]. After evaluating these tools (see section 2.5), *FitNesse* was selected as the most suitable test tool for Everett's purposes. FitNesse is an acceptance testing framework that is lightweight, open source and easy to use. See section 4.6 for more information on FitNesse.

1.5.2 Solution area: Identity & Access Governance

The strategy has been created and applied on the solution area of Identity & Access Governance projects (see section 2.2.2). This solution area has been chosen because Everett experienced a high demand for these projects, making it more likely that such a project would be available for the case study than with other solution areas. Furthermore, Identity & Access Governance projects are short term in particular (10 to 15 weeks).

1.5.3 Identity & Access Governance Software: SailPoint IdentityIQ

Designing a strategy for all products and suites that Everett uses in identity solutions projects would have made this research project far too comprehensive, so a representative product has been chosen to serve as reference, keeping in mind that the overall strategy should be generic enough to extrapolate it for

other products as well. *IdentityIQ* from *SailPoint* [11] was chosen as this representative product, as it is a well-known vendor software for Identity & Access Governance, and it is widely used at the time of this research project.

SailPoint IdentityIQ is a governance-based identity and access management suite [12]. It offers an identity warehouse where the identities are stored in a central repository, a role model, a policy model, an advanced risk model [13] and a workflow engine. Together, it enables the client to, for instance, create and manage roles, automate access certifications, automate changes based on lifecycle events (i.e., hiring, transferring, leaving), calculate access risks and define, detect and enforce policies. During the project, SailPoint IdentityIQ) becomes the center of all the main applications at the client's business for identity and access related information (see figure 1 on page 12).

1.6 Outline

This thesis is divided in 7 chapters, starting with this chapter which provides an introduction to the thesis. Chapter 2 gives background information of an earlier research project, providing preliminary knowledge for this thesis. Then, in chapter 3, the research method is described in depth. Next, a literature review is given in chapter 4, discussing important concepts and giving the proper literature background through the thesis. Chapter 5 then gives the high level strategy which was designed prior to the case study. Chapter 6 describes how this high level strategy is fine-tuned during a case study. Chapter 7 then gives the results of this thesis, discussing the final strategy, the application of the strategy, the evaluation of the strategy and an giving analysis of the results of this thesis. Chapter 8 discusses the thesis by giving the strengths, weaknesses and future work of this thesis. Finally, the overall conclusion can be found in chapter 9.

2 Background

This chapter is based on the author's earlier study, performed during the course 'Research Topics' [4], and offers background information for this thesis. The earlier study has been performed by a literature study, an observing case study at a project of Everett (including observations and personal interviews) and an analysis between theory and Everett's practice.

2.1 Chapter Summary

Everett's identity solutions projects are system integration projects that revolve around the scalable and timely management of users and their access to information and applications [3]. These projects are characterized as consisting of highly configured and customized third party software and being highly integrated, data-driven and short term (10-15 weeks) [4]. Identity solutions projects are performed at different clients at several sectors and may be implemented with various techniques and third party products [14].

The current system development process of Everett is based on the Scrum and Prince2 methodologies [15]. Everett already has Continuous Integration (revision control using GIT) and automated builds (using Maven or Ant) in place. Furthermore, Everett uses the Atlassian Software Stack (Bitbucket, JIRA, GreenHopper and Confluence) for revision control management, issue management, scrum project management and documentation [4].

The current test strategy of Everett consists of mostly manually and ad-hoc testing; tests are risk-driven; there are no formal test scenario's (except for user acceptance tests), negative test cases are seldom tested, testing is often performed with production data and there are no special test tools used in the project [4].

An analysis between the current strategy and the literature resulted in a set of improvement points [4]:

- Everett should be more test-oriented.
- A general test-strategy should be created.
- The handling of the system's high level of integration should be improved.
- Test automation with FitNesse should be introduced.
- Fictional data should be used instead of production data.

Based on the requirements given in section 1.2, FitNesse has been selected as a promising test automation tool for Everett's projects [4].

2.2 Identity Solutions Projects

Identity solutions revolves around the scalable and timely management of users and their access to information and applications. Everett delivers solutions with which organizations have, even across organizational boundaries, means to [3]:

- Reduce the operational and development costs of IT.
- Increase security.
- Comply with policies and regulations.

- Simplify business processes.
- Personalize services.

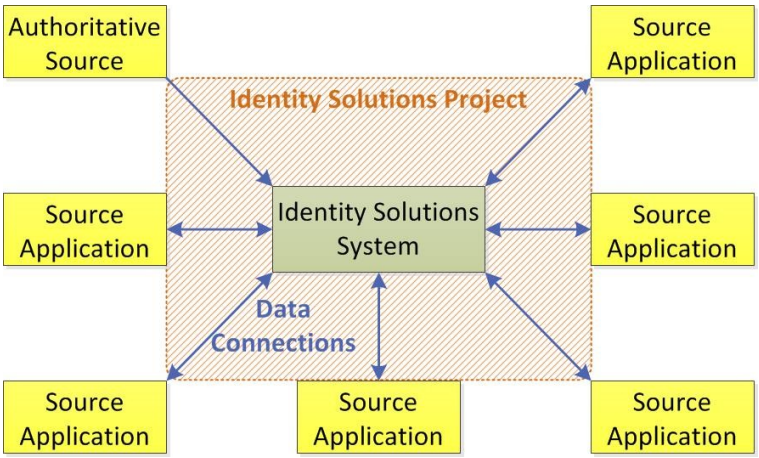


Figure 1: An architectural overview of an identity solutions project

Figure 1 shows an example of the architectural overview of an identity solutions project: An identity solutions system is introduced, configured and connected with several applications of the client and an authoritative source for identities (for instance, a human resource application).

The identity solutions system can perform several actions with identities, their access to information and source applications. The result is integrated management and use of identity information regarding employees, partners, suppliers and other stakeholders to support a complete service chain. Examples of identity solutions projects can be found from section 2.2.1 to 2.2.6.

Identity Solutions projects (ISP) are often realized in a short period of time (10 to 15 weeks). Everett consults, designs, implements and supports [16] identity solutions for their clients and uses various products and technologies in order to create the best fit for their clients requirements. The various products that act as an identity solutions system include products and suites from vendors like *ForgeRock*, *iWelcome*, *Microsoft*, *NetIQ*, *Oracle*, *RM5*, *SailPoint* and various open source products [14].

Identity solutions is a name that Everett gives to their defined set of several solution areas [3]. These solution areas are defined as groups of functionalities which are present in identity solutions projects. Multiple solution areas can be combined in one identity solutions project and there is some overlap between them. The following solution areas are defined:

- *Identity management*
- *Identity & access governance*
- *Access management*
- *Identity federation*
- *Identity cloud solutions*
- *Authentication*

These solution areas are explained in the following subsections.

2.2.1 Identity Management

Identity management [17] revolves around managing the lifecycle of identities and the subsequent relevant impact on their access to applications and services. It enables the company to give a new user quickly and automatically access to systems that match his relation to the company, using the same account for each system. Identity management can include self-service (manager can assign rights to its employees through an access request portal) and provisioning (automatic process that executes the changes in access rights the ICT landscape).

Example When the role of the employee changes (e.g., via promotion), he automatically gets the correct access for his new role. This includes withdrawing the access he had for his previous role but does not need for his new role. Furthermore, when the employee leaves the company, all access rights are automatically removed. Secondly, the manager can inspect some company data (name, personnel number and function) and access data of the employees that he supervises.

2.2.2 Identity & Access Governance

Identity & Access Governance [18] revolves around being in control of access rights to the information systems and the ability to demonstrate and prove it. It enables companies to comply with the regulations on access control set by regulatory bodies and gives these companies the ability to prove it as well.

Example Identity & access governance can enable companies to comply with the report from the Basil Committee on Banking Supervision [19], which states that the e-banking security process should include, among others, sufficient logical controls and monitoring processes to prevent unauthorized internal and external access to e-banking applications and databases.
In order to be in control of access rights and to be able to prove it, certification-cycles and reports are performed and generated periodically (for instance every quarter). In the certification-cycle, access rights of employees and accounts are verified by, for instance, the employee's manager or the system owner. During this verification, the manager can accept, revoke or redirect de decision to another employee. If the right is revoked, the identity solution system will take the appropriate actions: either by directly revoking the access via provisioning, by entering a ticket to the associated ticketing system or by sending an email to a specified address with the message to change the access right. Reports can be generated to provide an overview of information, for instance, describing which identities had access to which systems and information in a specified period of time.

2.2.3 Access Management

Access Management [20] revolves around the run-time evaluation and enforcement of what users are allowed to do when accessing a service. It provides a central mechanism to manage access of users, including comprehensive audit trails and important end-user functionality such as single

sign on. Access Management relies on authentication solutions to validate the identity of the users and relies on Identity & Access Governance solutions to determine authorizations.

Example When a user wants access to information (e.g. log in the system with customer data), access management techniques identifies the user (for instance by asking a username and password), and checks whether the user has the right authorizations to access the information (by verifying the credentials). Finally, the user gets access to the information (e.g. gets access to the system with customer data) when both steps are performed successfully. When the user does not have the correct rights, an error-message will be shown.

2.2.4 Identity Federation

Identity Federation [21] revolves around all processes and underlying technology which makes it possible to exchange identity data across organizational boundaries in a secure and controlled manner. It implements a comprehensive and robust architecture for Identity Federation, establishing a solution for integrated services in a supply chain.

Example Identity Federation enables secure collaboration across organizational boundaries, giving parties access to each other services. This can be between, for instance, suppliers and buyers: the buyers can access the stock information of suppliers and place orders directly.

2.2.5 Identity Cloud Solutions

Identity Cloud Solutions [22] revolves around integrating cloud applications, enabling one-click access to all applications, anywhere, anytime and from any device, while keeping identities safe. It delivers a single point of access to the company's public and private applications.

Example Identity Cloud Solutions enables companies to integrate and manage identities for cloud applications like Google, Salesfore and Office365 with their internal applications, letting their users access these cloud applications via the same portal as the company's internal applications, possibly with single sign on as well.

2.2.6 Authentication

Authentication [23] revolves around the process whereby a user’s claim to an identity is verified. It is one of the cornerstones of information security as it ensures both the traceability of actions performed within a system and that an identity is what its claims to be. Authentication is performed by validating a user’s credentials for the claimed identity. These credentials can be something that someone knows, possesses, is or a combination of these. Everett helps selecting and implementing the right authentication scheme to provide the required security level balanced with other business drivers.

Example	Authentication can verify a user by something that someone knows (e.g., a password or PIN code), something that someone possesses (e.g., a key, token or phone) or something that someone is (e.g., a fingerprint-scan or iris-scan) or a combination of these. With these authentication techniques, a user can ensure his identity in order to get the access rights that are assigned to this identity.

2.3 Current System Development Strategy

Everett develops according to the *Scrum* methodology [24]. Scrum works with iterations called *sprints*, typically lasting between two and four weeks. A Sprint is a fixed period of time for developing functionality as part of a product release (final product). Each Sprint will need to deliver some form of business value, adding on to the previous Sprints [15]. Each sprint starts with a planning and ends with a review. In between, *user stories* are implemented and tested. A user story is a software system feature specified by the customer in everyday business language. An example of a user story is: "As a customer, I want to be able to login on the site in order to see my purchase history". Working with the Scrum methodology helps the project in continuously delivering working software to the client and keep on moving forward to success.

Everett combines the Scrum methodology with the project management methodology Prince2. Prince2 provides a control and governance framework that aligns with the stakeholders, business and budget owners and their project organization [15]. Figure 2 shows how Everett combined the stages of Prince2 with the Scrum development methodology. Prince2 identifies several stages of the project (shown in yellow), such as directing, startup, initiation, planning and so on. Everett has added scrum (shown in orange) to this diagram, which shows that the product delivery is done with sprints and a daily standup (i.e., the 24h-cycle), that a sprint starts with planning and ends with a demo where after a new sprint starts.

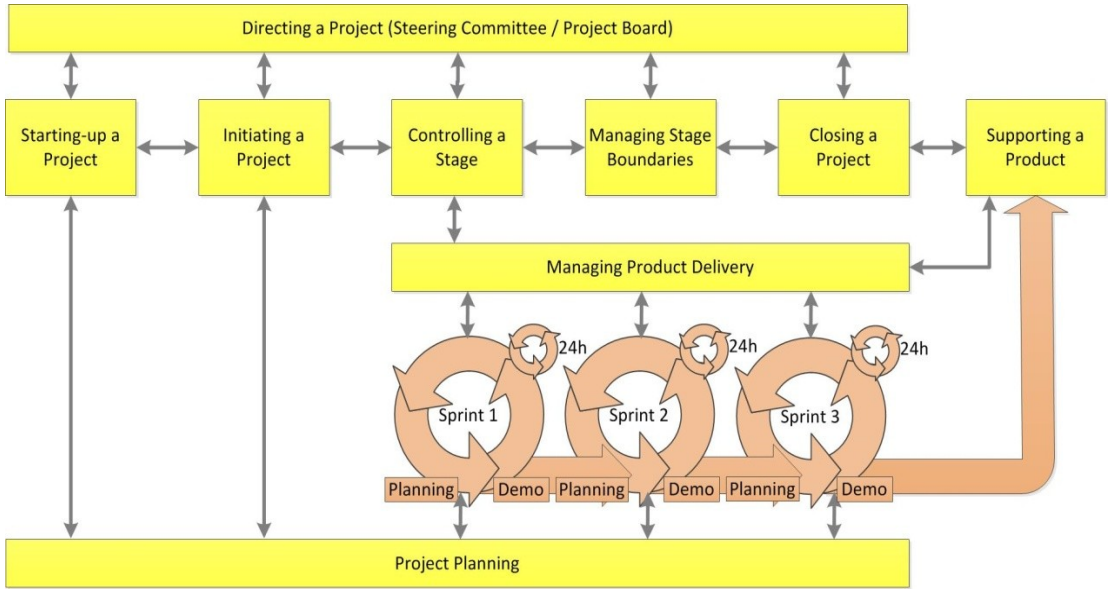


Figure 2: Involve: combining Prince2 (yellow) with Scrum (orange)[15]

As part of the development environment, Everett uses the *Atlassian* software stack [25] in their projects; *Bitbucket* combined with Git [26] for revision control (integrating code and manage the different versions of this code), *JIRA and GreenHopper* for issue and scrum project management (managing user stories and issues to work on), and *Confluence* for documentation (wiki).

Furthermore, Everett uses mostly *Maven* [27] and *Ant* [28] as automated build tools (automatic compiling and building the projects code into an application that can be run). With these tools, Everett has build scripts that can be run via, for instance, a command line in order to build the project.

2.4 Previous Test Strategy

When using scrum, a project has a *definition of done*, which specifies the criteria of when a task in the project is considered done. An example/standard definition of done of Everett has three specified test activities, performed by three different parties/individuals:

- Developer: The developer tests his implementation of the user story.
- Review: The implementation of the user story needs to be reviewed by another team member.
- Acceptance testing: When the sprint is finished, the client tests the implementation of the user story, as part of acceptance testing.

Earlier research [4] notices that, except from the statements in the definition of done, there was no clear or standard strategy defined at Everett, so an observing case study was held in order to investigate the test process in practice. In this case study, team members of a project were interviewed on the test process. This project was also an Identity & Access Governance project, also using SailPoint IdentityIQ. The case study showed that:

- Most tests are performed manually without special test tools or test scripts; the developer checks his solution with a couple of debug statements and visually checking the output based on the requirements and the user story where the developer is working on.
- Tests are risk-based, meaning that for standard solutions, only the configurations are checked. For instance, for a standard connection, the hostname, logs and whether or not the data is received are checked visually, but there is no deep test for every detail. Negative (i.e., test cases that should cause an error or failure) test cases probably seldom or not performed by the developers.
- Tests are performed with production data, because this whole project revolves around this data. For instance, with production data it is possible to check whether or not rights have meaningful names, roles are representative and business rules are specified correctly. Furthermore it gives insight what the real effects of this project will be in production, which can only be assured with production-like data. Since there are guidelines and laws stating production data should not be used in test environments [29, 30, 31, 32], certain agreements are made before using production data. For instance, the data is only available on the test environment and will not be available to the cloud.

- Unit tests are seldom used, since IdentityIQ uses a script language called BeanShell which doesn't support unit tests. However, it is possible to let BeanShell call Java-functions and perform unit tests on those Java-functions.
- Instead of unit tests, integration tests are performed. Ideally end-to-end, although it is almost never possible to obtain full test coverage of the whole software chain. The missing parts of an end-to-end test are then simulated by mocking.

During interviews in an observing case study in earlier research [4] several improvement points have been mentioned as well: According to the Junior Engineer, testing was started too late, which gives a risk that problems are discovered in a late stage where a lot of work may be redone. The project manager stated that it probably would help to add someone to the project that monitors the test process and helps with defining correct test cases. Secondly, the project manager stated that it might help to formulize test cases so they can be reviewed and correlated to requirements and risks, introducing transparency to the client. These test cases are then probably reusable for other projects as well.

2.4.1 Improvement Points

The earlier research project [4] had performed an analysis over the literature combined with the observed strategy, identifying gaps between the applied practice and theory. These gaps were presented as improvements over the current strategy:

- More focus on testing should be created throughout the company in order to improve the test process. This focus is needed in order to investigate and apply the other improvements mentioned below.
- A new test strategy should be created and standardized to improve the test process, making testing more structured, documented and well thought out.
- Decompose highly-integrated systems into manageable and testable units to deal with the system's high level of integration. Mocking should be investigated in detail in order to help with this decomposition.
- Test automation should be introduced. In order to do this, a strategy should be designed that uses FitNesse.
- Techniques like data anonymization and generation should be investigated, including proper tools which can assist in applying these techniques for identity solution projects.

2.5 Test Automation Tool Selection

As said in the introduction, several automation steps are needed in order to set up continuous delivery: configuration management, automated build and deploy scripts, automated tests and a continuous delivery framework. As statded in section 2.3, Everett has the configuration management and automated builds already in place.

In the earlier research project [4], FitNesse has been chosen as automated acceptance test tool, since this choice will have a huge impact on the strategy.

Jenkins was chosen during the case study of the thesis as continuous delivery framework (see section 6.2.1). The rest of this section will elaborate on the choice for FitNesse.

There are several tools available for the purpose of test automation, such as web browser automation tools like *Selenium* [5] and *Watir* [6] and the test automation tools *FitNesse* [7], *GreenPepper* [8], *Cucumber* [9] and *Root Framework* [10]. These tools are compared with each other based on the requirements in section 1.2.

A short list of the functionalities and usage of the possible tools is given:

- Web browser automation tools like Selenium and Watir perform tests on the web GUI of the systems under test. These tools want several interactions with the web browser and validations (i.e. the current pane must contain the word "x") as input and give true or false for each validations. Selenium is based on multiple programming languages: Java, C#, Groovy, Perl, Php, Python and Ruby whereas Watir is a Ruby library. Both tools support multiple browsers and Selenium also has the feature to record and playback tests [5, 6].
- FitNesse is an open source wiki and acceptance testing framework, enabling teams to collaboratively define acceptance tests. This tool expects test tables in a wiki format, which interact with Java programs (called fixtures) that communicate with the system under test. When the acceptance tests are run, the results are given in the wiki. It can be used with one of the two frameworks as basis: Framework for integrated tests (FIT) [33] or Simple List Invocation Method (Slim) [34]. FitNesse has plug-ins for Maven and Git and can be connected with the continuous integration tool: Jenkins [7].
- GreenPepper is very similar to FitNesse, although it is not open source and does only supports the FIT framework. It has build-in support to connect it with the Atlassian software stack, Maven and some IDE's [8].
- Cucumber is an open source framework where users can create behavior and scenarios in plain text and write a definition in Ruby that interprets this plain text and calls the system under test [9].
- Robot Framework is an open source framework where users create tests in a tabular test data syntax. It uses keywords provided by the test libraries (written in Python or Java) to interact with the system under test. The results are given in HTML format and XML output [10].

Table 1 gives a comparison table between the tools and shows how well they score on the requirements for our strategy given in section 1.2, including some clarification for the score. The table shows that FitNesse is the most suitable tool for our purposes by having the most plusses and benefits and the least minuses and disadvantages. Therefore, FitNesse has been chosen as test tool for the strategy. More detailed information on FitNesse is be given in chapter 4.6.

req. → / tools ↓	Efficacy	Flexibility	Implementation time	Cost-effectiveness	Transferability
Web browser automation tools	- tests higher level than needed	-- although multiple browsers, only tests web browser apps	++ can record tests, Java	?- depends on implementation time but has maintenance when GUI changes	+- open source but maintenance when GUI changes
FitNesse	+ can test multiple test levels (e.g., unit, integration and acceptance tests) wiki to create and show test cases and its results	+ can test all apps that can be called via Java (optionally via other languages with language plug-ins)	++ separation between test definition (wiki tables) and test coding easy to deploy	? depends on implementation time	+ open source
GreenPepper	+ same as FitNesse	+ same as FitNesse	++ mostly same as FitNesse, however: it connects to the Atlassian software stack that Everett uses and does not support the slim framework	? depends on implementation time	- closed source
Cucumber	+ multiple test levels	+- can test all apps that can be called via ruby	- separation between test definition (plain text) and test coding, but need to learn ruby	? depends on implementation time	+ open source
Robot Framework	+- multiple test levels reports in html and xml format	+ can test all apps that can be called via python and Java	+- separation between test definition (tabular data syntax) and test coding	? depends on implementation time	+ open source

Table 1: Test automation tool comparison table

3 Research Method

This chapter explains by which method the research questions from section 1.3 have been answered by this research project. The main research question and its underlining research questions are listed here as reminder:

- *How can continuous delivery and test automation with FitNesse be introduced in system integration projects?*
  1. *What is a good strategy to introduce continuous delivery and test automation with FitNesse?*
  2. *What is are the costs and benefits of applying the strategy?*

3.1 Chapter Summary

The first underlining research question (i.e., what is a good strategy) has been answered by creating a high level strategy for test automation and continuous delivery based on literature by selecting promising combinations of methods, guidelines and tools and creating tutorials when needed. When the high level strategy is created, it will be applied, evaluated, supplemented and improved during a case study which is discussed below.

The second underlining research question (i.e., what are the costs and benefits) has been answered by evaluating the strategy in a case study. The case study approach is chosen to investigate the validity of the strategy in the real-life context of an identity solutions project in a qualitative manner. The case study followed an iterative research pattern, which has intermediate evaluations and changes in order to design the strategy in steps and improve it along the way. The case study is performed in five iterations, each during two weeks. Data is gathered from documents, observations, surveys and personal interviews.

3.2 High Level Strategy Design

The high level strategy design has been created prior to the case study in order to have a well-thought-out solution basis at the beginning. Much details were then yet to be determined. This high level strategy consist of tool-selections, tutorials approaches and guidelines which are documented in informative documents. The high level strategy design is listed in chapter 5. The strategy has been given more details and is adjusted and improved during the case study.

3.3 Case Study

The case study's project needed to be a project at a client of Everett and consists of implementing, testing, evaluating and improving the strategy design.

The case study's main goal is to determine the validity of the designed strategy with as secondary goal to add details, improve and adjust the strategy. With the case study approach, the validity of the strategy has been investigated in the real-life context of an identity solutions project in a qualitative manner. Data is gathered from observations, surveys, interviews and documents.

A case study is a qualitative research method. Qualitative methods can help to identify problems and improve the strategy design during its development [35]. According to Yin [36], case study research is appropriate for investigating a

phenomenon in its real-life context, and for answering how and why questions when the investigator has little control over the events.

3.3.1 The Case Study's Project

The case study's project has been performed at a client of Everett where an Identity & Access Governance project takes place. This project uses the SailPoint IdentityIQ software and was the second phase of a project at this client.

The first phase introduced SailPoint IdentityIQ at the client, migrated the management of role models from Excel to IdentityIQ, integrated six applications with IdentityIQ, and finally introduced periodic certifications, periodic verifications of access rights and periodic reports on risks and policy violations.

The second phase, which is project of the case study, will extend the first phase by using SailPoint IdentityIQ's LifeCycleManager [37] as an application portal with lifecycle events, introducing provisioning and connecting more applications for certifications. Phase 2 will be conducted with the same team members as in phase 1 with four people of Everett and three people of the client. The project is conducted with sprints of two weeks whereas Everett consults three days in a week.

3.3.2 Case Study Approaches

Iterative Research Pattern

The case study had been used an iterative research pattern as described by Pratt [38]. This model allows the design of the strategy to be constructed in steps, beginning with a basic strategy design which will be applied, evaluated and adjusted and in several iterations. This differs from the non-iterative method, which requires a final and complete strategy before validation. To create such a final and complete strategy at once is difficult to achieve without practical experience in both performing identity solutions projects and in creating and applying test strategies. Because of this, the iterative research pattern suits the case study better.

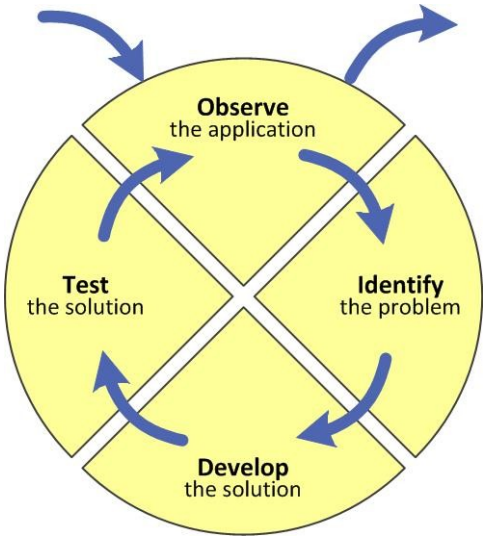


Figure 3: Iterative research pattern [38]

As shown in figure 3, the iterative research pattern consists of four primary steps with a cyclic relationship: observe the application, identify problems, develop the solution and test the solution. The pattern starts and ends with the observe step. The pattern should be used with several iterations. If not, the pattern degenerates to a waterfall development model [38].

Prior to the case study, three steps already have been performed: observing and identifying the problem is done in a previous research project (see chapter 2) and the development of a solution is done by creating the high level strategy (see

chapter 5). Because of this, the case study has been started halfway of the first iteration at the test-phase.

Interpretive perspective

There are several philosophical perspectives on how to conduct valid qualitative research, including the commonly known positive and interpretive perspective. The positive perspective assumes that that the reality is objectively given and can be gathered objectively, whereas the interpretive perspective assumes that the reality is subjective and can only be gathered through social constructions [39, 40, 41].

The effects of the strategy is not well or at least not completely measurable through only objective data (e.g., the amount of errors found). Therefore, the subjective opinions (e.g., how does a participant experience the strategy) have been considered of great value. resulting in the choice of the interpretive perspective.

Using the interpretive perspective, the strategy is evaluated through the opinions of the team members (what did they find difficult, what did they want to have improved, what worked well, etc). This gives the opportunity to adjust the strategy during the case study, based on these opinions.

In order to perform the interpretive approach properly, the case study has been following the set of principles for conducting and evaluating interpretive field studies in information systems given by Klein and Myers [42] (see appendix A for their summary of these principles).

Inquiry from the inside

Next to perspectives, there are multiple inquiry modes; inquiry from the inside and inquiry from the outside. With inquiry from the outside, the researcher is totally detached from the organizational setting of the investigation, where with inquiry from the inside, the researcher is personally involved in the investigation [43, 44].

The case study has been performed as an inquiry from the inside, as the researcher has been a part of the team and helped the team implementing the new strategy. It has been chosen above the inquiry from the outside for several reasons. First of all, more experience and knowledge on practical issues and opportunities of the strategy can be gathered with inquiry from the inside, which helps with making more founded choices when adjusting the strategy. Furthermore, the team members of the case study had limited amount of time to apply the strategy, making it useful to have some tests already in place. These tests were then used as examples and reference material when evaluating the strategy, its capabilities and its limits. Finally, this approach enabled the case study's team members to ask for more clarification, details or examples about the strategy when necessary.

3.3.3 Case Study Detailed Design

Iterations

With the chosen project and approaches, a detailed case study design has been made. First of all, a time scope of the iterations has been defined. Since the project already works with sprint as development cycles, one iteration matches one sprint. In total, five sprints have been used for the case study. It is decided to held five sprints in order to gather a proper amount of feedback, enough room for improvements and a duration that is long enough to be able to perform a final validation, but not longer in order to stay in schedule of the thesis. In total, with an occupation of three days a week, the case study had been performed in thirty workdays, spread over ten weeks.

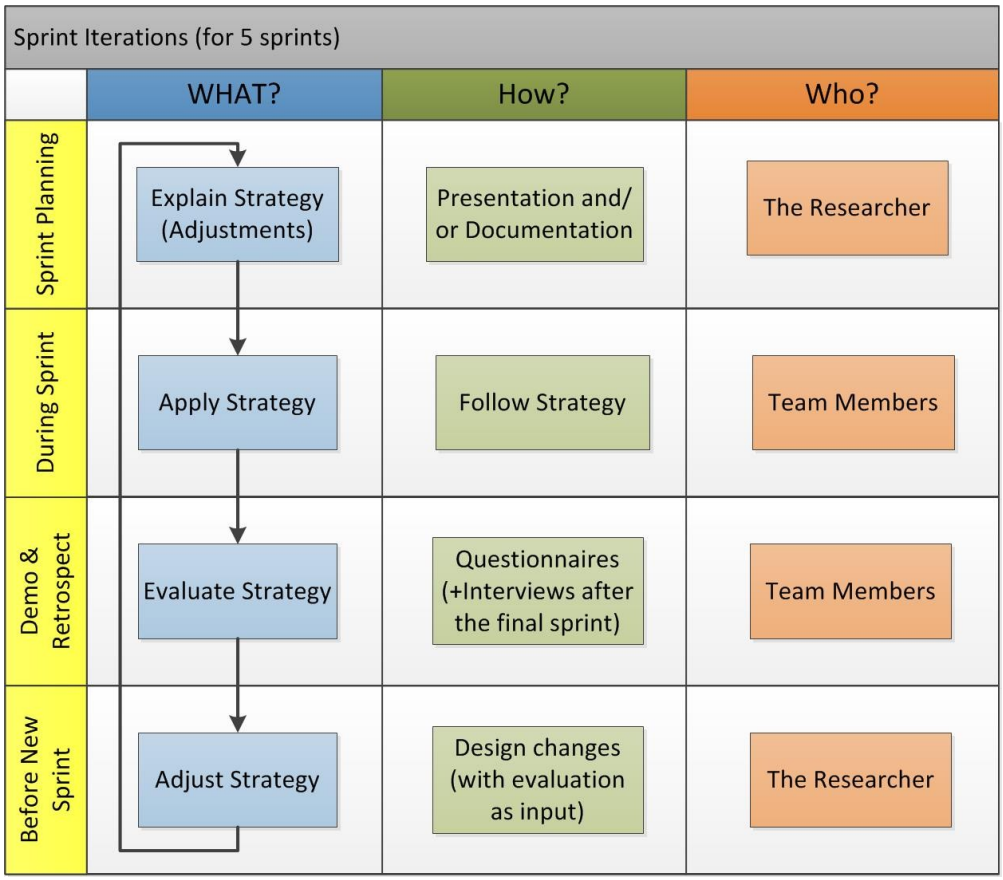


Figure 4: Case study detailed design

Figure 4 gives an overview on which activities (blue) took place at which moment of the iteration/sprint (yellow), how this is done (green) and by who (orange):

- Explain strategy: The sprint planning has been the ideal time to explain strategy (adjustments), since the sprint planning marks a new sprint and is used as a new start.
- Apply strategy: Naturally, the application of the strategy has been done during the sprint, since then the user stories will be implemented and tested.
- Evaluate strategy: The strategy has been evaluated during the retrospective, since the retrospective is the place where normally the sprint is evaluated as well, having everyone already in an evaluating

mood. These evaluations will be hold by presenting everyone a survey, which was filled in on the spot.
In the week after the final iteration, in-depth interviews with the team members have been hold for each team member separately.

- Adjust strategy: The strategy was adjusted before the new sprint, so that changes can be communicated before the new sprint. Because sprints end at Thursdays and start again at Mondays, it leaves the Fridays open to design the adjustments based on the evaluations on Thursdays. Adjustments that took more time were designed during the following sprint and introduced in the sprint after that.

The activities shown in figure 4 matches the iterative pattern of figure 3 on page 21, in a way that is shown in table 2: The application has been observed by observing how the strategy is applied, the problem has been identified by evaluating the strategy, the solution has been developed by adjusting the strategy, then the strategy adjustments has been explained, where after the solution has been tested by applying the adjusted strategy.

Step of the iterative research pattern (figure 3)		Steps of the case study detailed design (figure 4)
Observe the application	<->	Apply Strategy
Identify the problem	<->	Evaluate Strategy
Develop the solution	<->	Adjust Strategy
<inbetween>	<->	Explain Strategy
Test the solution	<->	Apply Strategy

Table 2: matching steps of the iterative research pattern with steps of the case study detailed design

During the sprint, I (i.e., the researcher) was present on location: observing the project, supporting team members with the strategy, designing adjustments and implementing some example test cases.

Surveys

The survey at the end of each iteration has been used to gather information on how the team members experience the new strategy: what is clear, what could be improved, which problems did occur, etc. The main goal of the survey has been to determine which parts of the strategy worked well, which parts were unclear, which parts were needed to be adjusted and what was missing. Secondly, it has been used to identify the costs and benefits of the strategy, including outliers and trends over the iterations.

The survey has been chosen instead of a verbal group setting to let the participants not influence each other. Furthermore, it has been performed on paper instead of on a digital medium, so it could be filled in directly at the retrospective and it would not have been be postponed or forgotten.

The survey contained mainly questions with answers that use a five-point scale and some room for explanation if necessary. Scale-questions can be filled in quickly and are suitable for comparison between sprints. Additionally there

were open questions for those where I couldn't provide pre-defined answers, but the answers could be very valuable when improving the strategy.

The complete survey is enclosed in appendix B. Below states what each question of the survey contributes to the thesis.

- The first question asked about the amount of experience from the team members on Sailpoint IdentityIQ, FitNesse, test-driven development and test automating. The answers on this question can be used in order to determine with how much experience the other questions are answered and possibly show a growth in expertise during the case study.
- The second question asked how well the strategy parts lend their selves to do their task and to what extent the tutorial and explanations were good to follow and apply. These questions are used to evaluate and adjust the strategy during time.
- The third question asked the amount of hours spend on test-activities in order to give rough estimations for the time it costs to apply the strategy
- The fourth question asks whether problems are encountered, how restrictive these problems are and if they are solved (and how). This question is used for strategy improvements in order to determine limitations for the strategy and possible improve the strategy by fixing the problem in a next iteration.
- The fifth question asked how the strategy effects several aspects of the project by comparing these effects between the first phase with the previous strategy and the second phase with the new designed strategy and is mainly used for evaluation purposes.
- The sixth and seventh questions asked which part of the strategy could be reused in which manner and/or to which extent in order to determine the reusability of this strategy in other projects.
- Finally, there was room for comments, tips, ideas, complaints or pitfalls of the strategy.

Interviews

The final interviews after the last sprint have the goal to gather an overall evaluation of the strategy. This evaluation is done by comparing my strategy with the strategy used in the first phase of this project and comparing my strategy with the goals and requirements given in section 1.2. All interviews are recorded, transcribed and send back to the participant in order to be able to verify the statements done in the interviews.

The interview mainly contains open questions, asking the participants to evaluate the strategy as a whole. The interview gives the opportunity to respond on their answers, asking for more clarification, details and examples. The interviews helped to achieve a qualitative and complete evaluation of the strategy.

The complete list of interview questions is enclosed in appendix C. Below states what each question of the survey contributes to the thesis.

- In the first part of the interview, the team members have been asked to give positive and negative aspects of the strategy. This question gives the

team member the change to give his opinions without further influences or steering towards an answer. The answers given on this question were a guidance for other questions as well, so the interviewer knows where it could ask for more information or clarification.
- The second part of the interview consists of questions that asked about the costs, benefits, effects and specific evaluations of the strategy. The answers given on this question determines mainly how the strategy performed and is evaluated.
- The third part of the interview consists of questions that asked to which extent the goals of this thesis has been achieved in order to use that for evaluations and future work.

3.4 Evaluation

The strategy was evaluated through analyzing the results of surveys and interviews.

Surveys were summarized by first grouping answers on questions. For the questions that have a scale as answer, the answers are grouped by survey iteration, calculating the averages per iteration (with min and max values) and the overall average (with min and max values), and creating a plot from this, both with individual values and average values. For the open questions, all individual answers are listed, sometimes grouped per iteration.

Interviews were summarized by grouping answers on questions or subjects, listing the opinions of all individuals on that question or subject.

From both summaries (of the surveys and the interviews), the results were analyzed by linking answers on related subjects and questions, relate them to each other, to the application of the strategy and/or to the background of the individuals in order to draw conclusions.

4 Literature Review

This chapter provides theoretical background and the state of the techniques on important concepts of the research project. The following concepts are discussed:

- Continuous delivery
- Test models
- Test-driven development
- Test automation
- FitNesse

4.1 Chapter Summary

Continuous delivery is used in software development to automate and improve the process of software delivery by using a deployment pipeline that consists of continuously integrating, building, testing and releasing software [45, 46].

Test models describe test activities with correlation to development activities. The v-model shows that testing can (and should) start at the beginning of the project. An improved version of the v-model [47], called the advanced v-model [48], is designed to reflect the relationship between the development activities, test activities and maintenance activities. Another improved version of the v-model, called the w-model [47], is designed to define more clearly when which test activity starts, what the connections are between various test stages, and it shows the link between the tasks of testing, debugging and changing during test phases.

Test-driven development (TDD) is a development and testing practice from the agile software movement where tests are written before coding in small iterations existing of: write a failing test, make the test pass, refactor [49].

Test automation is the use of a mechanism for running test cases without a tester, which can improve the quality of testing but also requires an investment that should repay itself [50, 51].

FitNesse is a test automation tool that is used in the strategy design of this thesis. It is a lightweight, open source framework that enables teams to collaboratively define acceptance tests, run those tests and see the results. FitNesse tests can be used very early in the project and tests can be written by both technical and non-technical stakeholders [7].

4.2 Continuous Delivery

Continuous delivery is used in software development to automate and improve the process of software delivery by using a deployment pipeline that consists of continuously integrating, building, testing and releasing software [45, 46].

4.2.1 The Deployment Pipeline

The Continuous delivery deployment pipeline [45, 46] is shown in figure 5:

- First, the delivery team delivers commits to the version control, integrating code in one central place.
- The update in the version control software triggers the build and unit tests, which verifies that the system compiles and passes a suite of automated unit tests.
- When the build and unit tests pass, a series of automated acceptance tests are triggered, which tests whether the system works at the functional and nonfunctional level.
- When the automated acceptance tests also succeed, the manual user acceptance tests can start, testing whether or not the system is usable and fulfills its requirements.
- When the manual tests succeed, the delivery team will get this success as feedback and the software can be released.
- When one of the tests fails, the delivery team will get this fail as feedback.

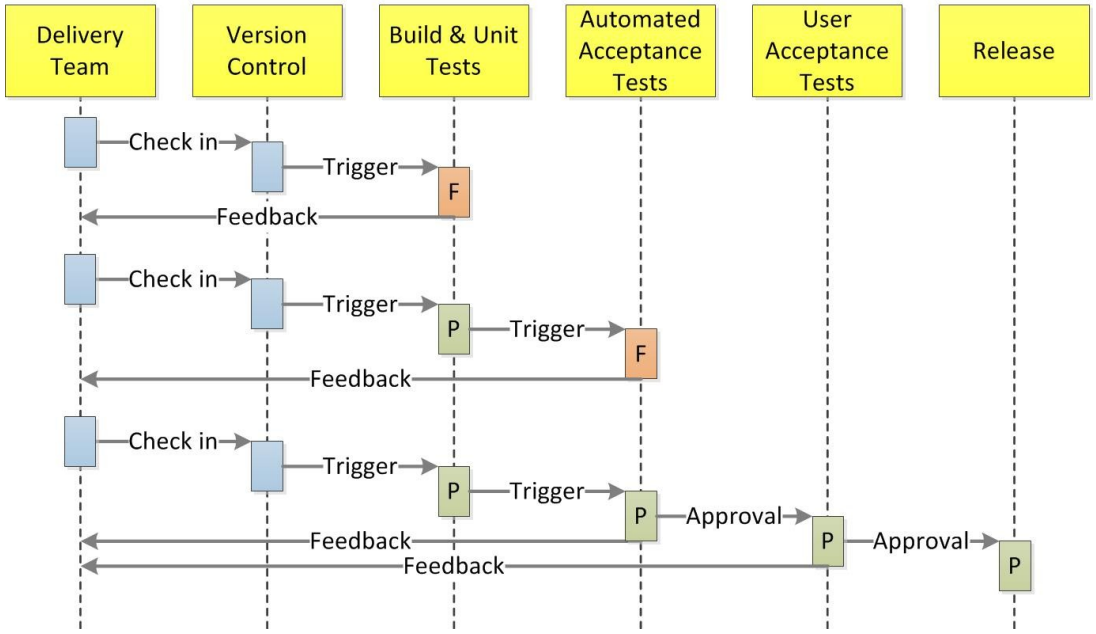


Figure 5: Changes moving through the deployment pipeline [46]

4.2.2 Prerequisites of Introducing Continuous Delivery

The deployment pipeline depends on having some foundations in place [46]:

- Good configuration management.
- Automated scripts for building and deploying your application.
- Automated tests to prove that your application will deliver value to its users.

With these foundations, a continuous delivery framework can be set-up in order to perform the triggers and feedback steps shown in figure 5.

Successful continuous delivery requires discipline, such as ensuring that only changes that passed the automated build, test, and deployment get released.

4.2.3 Supporting Tools

Several tools exists which support steps of the continuous delivery pattern. Below, some tools are listed as starting point for the automated steps of figure 5:

- Version control software like *Git* [26] and *Subversion* [52] for configuration management.
- Build tools like *Maven* [27] and *Ant* [28] for automated builds and tools like the *xUnit frameworks* [53] for unit tests.
- Test automation tools like *FitNesse* [7], *GreenPepper* [8], *Cucumber* [9] and *Root Framework* [10] and - for automating web browsers - tools like *Selenium* [5] and *Watir* [6] for automated acceptance tests.
- Continuous Integration tools like *Jenkins* [54] and *Bamboo* [55], for managing the overall process, triggers and feedback steps of continuous delivery. These tools offer to perform and monitor external jobs that build and test software projects continuously.

4.3 Test Models

There are several test models that describe test activities in relation to development activities. The V-Model is the most commonly known. Later on, the Advanced V-Model and W-Model were introduced as an improved version of the V-Model.

4.3.1 V-Model

The v-model describes the graphical arrangement of individual software development phases, connecting development and test activities in different abstraction levels. The V points to both the form of the graphical representation shown in figure 6 and to the terms *verification* and *validation* [47].

The v-model shows that testing can (and should) start at the beginning of the project [48]. In figure 6, all grey arrows have the meaning of *based on*. For example, acceptance testing is based on the requirements, coding is based on the detailed design, and so on. The purpose of the v-model is to improve efficiency and effectiveness of software development- and maintenance activities by connecting development and test activities.

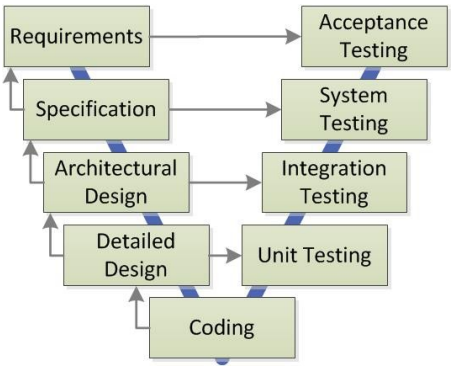


Figure 6: The v-model[47]

4.3.2 Advanced V-Model

An improved version of the v-model, called the advanced v-model, is designed to reflect the relationship between the development activities, test activities and maintenance test activities. It adds the maintenance line to the model, containing test cases, regression testing, security testing and deployment testing [48].

The advanced v-model shown in figure 7 has an extra line on the right compared with the v-model in figure 6. The figure illustrates that, when the test activity in the middle line is done, the test activity on the right of that activity needs to be performed afterwards (perform test cases after unit testing, perform regression testing after integration testing, and so on.) Again, all grey arrows have the meaning of *based on*.

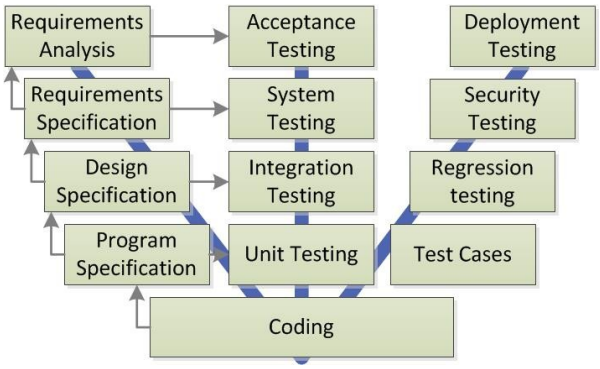


Figure 7: The advanced v-model [48]

4.3.3 W-Model

Another improved version of the v-model, called the w-model, is designed to define more clearly which test activity initiates which other test activities, what the connections are between various test stages, and what the link between the tasks of testing, debugging and changing during test phases are [47]. Figure 8 illustrates this w-model. If you compare the w-model in figure 8 to the v-model in figure 6, two lines are added:

- The second line from the left. This line states test should be planned and test activities should start early on.
- The line on the right. This line states that test activities lead to change activities by the discovery of faults and errors.

Again, all grey straight arrows have the meaning of *based on*, and as addition, the grey round arrows have the meaning of a *cycle* of debugging, changing and re-testing.

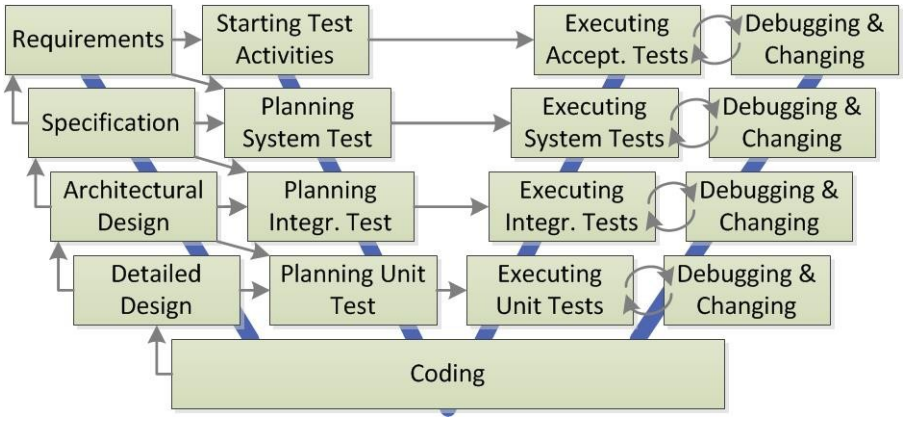


Figure 8: The w-model [47]

4.4 Test-Driven Development

In Agile Software Development, t*est-driven development (TDD)* emerged as a new development and testing practice from the agile software movement [56]. The idea of TDD is to write test cases before coding and work in small iterations to yield better quality and fewer defects in code. The principles of TDD are to test as early as possible, as often as possible, test just enough for the situation and to perform pair testing.

Figure 9 shows the most common development cycle of TDD [49]:

- Write an automated test case that defines a desired improvement or new function.
- Produce the minimum amount of code to pass the test.
- Refactor the new code to acceptable standards.
- (Repeat)

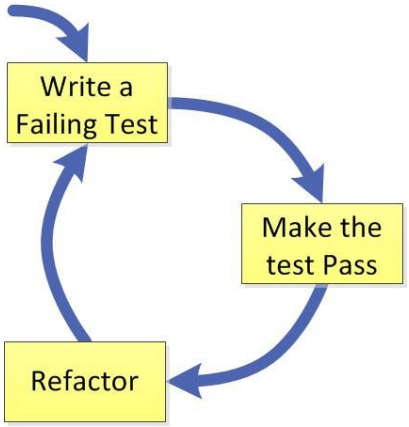


Figure 9: Test-driven development[56]

Using TDD as development style makes your code self-testing, in general more suitable for testing and it sets the focus on testing requirements instead of testing if the implementation is satisfied [57].

4.5 Test Automation

In order to achieve continuous delivery, tests should be automated. Manual testing can be described as a person initiating tests, interact with them, interpret, analyze and report the results. Automated testing is the use of a mechanism for running test cases without a tester [50].

4.5.1 Benefits

Test automation provides an improvement on the quality of testing; automated tests are formalized and can be run repeatedly for many times with minimal effort. The repeating property enables tests to run often, therefore finding errors more quickly than when manual testing is used, especially when modification in one component breaks another component [51]. Finding errors quickly saves time overall in the development processand can reduce overall development time by 8–15% [58, 59, 60].

4.5.2 What/When to Automate

The benefits mentioned above sound very promising, but test automation is not always profitable, since test automation comes with an investment: the costs of automating a GUI-test may be several times as expensive as a manual GUI test (although, relatively cheaper when a capture or replay tool is used). However, when automating compiler testing, automation is only a little more expensive than running manual tests (because both manual and automatic tests of compiles use the same syntax and it is fairly easy to put these syntax in a script) [51].

Automated tests have a lifecycle; they are run every time after the code changes until they need to be repaired or discarded. The investment of automating a test should repay itself before that test breaks.

To estimate the costs and benefits of automating, three questions should be asked [51]:

- How much more time does it take to automate the test instead of manually running it?

- How likely is it that the test dies(e.g., it needs maintenance or is thrown away)? Which events are likely to end it?
- During its lifetime, how likely is this test to find additional bugs (after the first run) and how does this balance with the costs of automation?

Note that there will always be a role for manual testing [51, 61]. First of all, it is the only way to sanity test the automation itself and secondly, some tests are not worth repeating due to high automation costs.

4.5.3 Tools

One of the key elements when automating tests for a system is knowing which tools can support test automation in the system's environment [50]. As stated in section 4.2.3, test automation tools like *FitNesse* [7], *GreenPepper* [8], *Cucumber* [9] and *Root Framework* [10], *Selenium* [5] and *Watir* [6] are a selection of automated solutions for acceptance tests.

Next to tools that automate acceptance tests, there are many tools for supporting other test activities as well, such as tools that automatically generate test cases or test data. Finding, selecting and adjusting these tools so that they are capable to work and deliver value in identity solutions projects will be a challenge on its own and out of the scope of this project due to time constraints and the choice of focusing on introducing continuous delivery.

4.5.4 Common Test Automation Problems

Pettichord illustrated several common problems that plague test automation projects [61], including that test automation does not get the focus it needs, that the goals of automation are not clear and that projects suffer from a lack of test experience from employees. Pettichord contends that test automation projects need to be run like other software development processes, needing test designs, source code management, user documentation etc. It is good to be aware of these problems to be able to early identify if these problems occur at your project.

4.6 FitNesse

As stated in section 1.5.1 and section 2.5, FitNesse will be used as test automation tool in the strategy. FitNesse [7] is a lightweight, open source framework that enables teams to collaboratively define acceptance tests, run those tests and see the results. FitNesse tests can be used very early in the project: it works well with test-driven development, where the tests are written first. FitNesse also offers an internal version control system that stores old versions of wiki pages automatically in ZIP-files as a backup [62].

With FitNesse, tests are specified in the wiki through test tables. This is done in a way that is user-friendly for both technical and non-technical stakeholders. The idea is that all stakeholders contribute on creating test tables in the wiki, since it represents requirements as a verifiable and executable table. With this, FitNesse can help with specifying and clarifying textual requirements because it forces stakeholders to come up with specific examples and think of specific exceptions.

4.6.1 Components

A simplified version of the FitNesse architecture, is given in figure 10. This figure shows only the parts that need to be created when using FitNesse: test cases, fixtures and the system under test [63]. Tests are specified in the wiki, from this wiki, the corresponding fixtures are called, which in their turn perform calls to the system under test and report the results back to the fixture, back to the wiki.

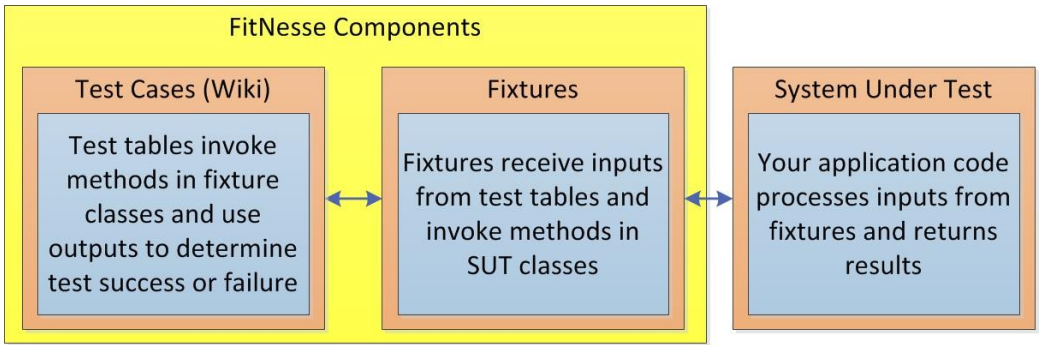


Figure 10: FitNesse components and the system under test (based on [64] and [63])

Wiki

The wiki has three types of pages: static, test and suite pages. Static pages are simple text pages, test pages are executable pages which contain test tables and suite pages are collections of test pages or other suites in order to group en order test pages. A screenshot of a test page with a script table is shown in figure 11. FitNesse stores the wiki as folders and plain text [62]. The wiki-pages can be run manually (via the Wiki) or automatically (via Ant, Maven, JUnit, REST-commands) by anyone with web access to the server, as frequently as required.

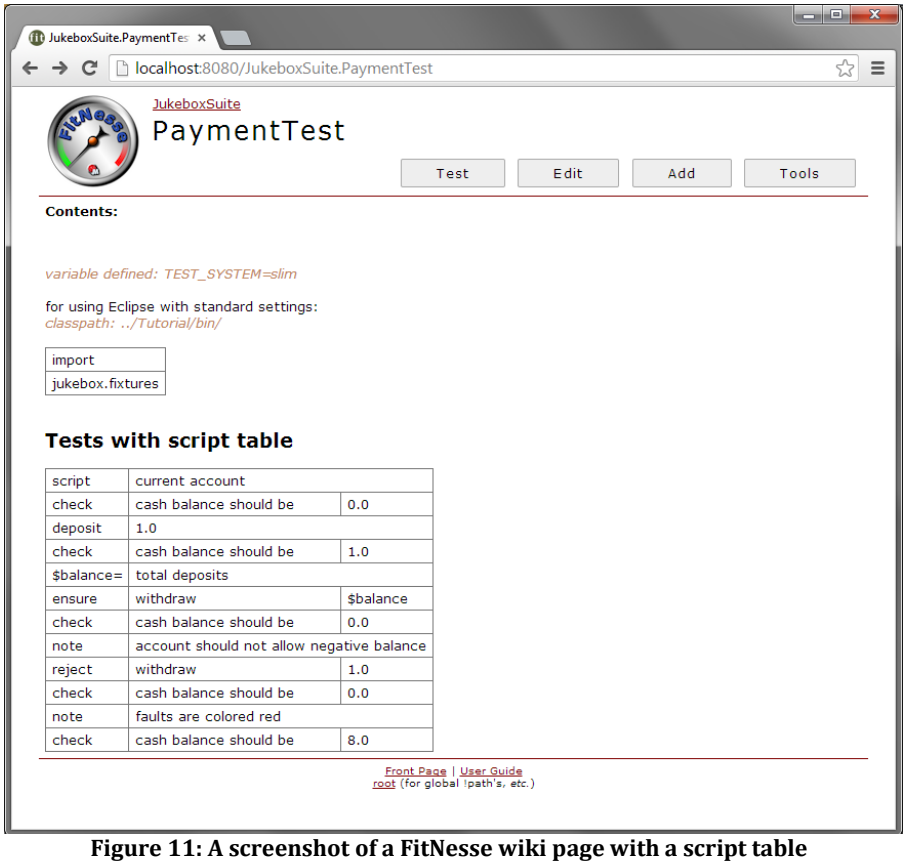


Figure 11: A screenshot of a FitNesse wiki page with a script table

Fixtures

The test tables in the wiki are associated with tests programs, called fixtures. These fixtures take the input data from the wiki, test the system under test and delegate the output back to the wiki again. FitNesse then compares the actual outcome with the expected outcome and reports the results by color-coding the table rows in the wiki: green for success and red for failure and, when a test fails, giving the errors and differences between what was expected and what was received from the system under test (SUT). Figure 12 shows the results after running the test page of figure 11. As you can see in figure 12, the test found an error (color coded with red) which is in this case due to a bad value in the script table.

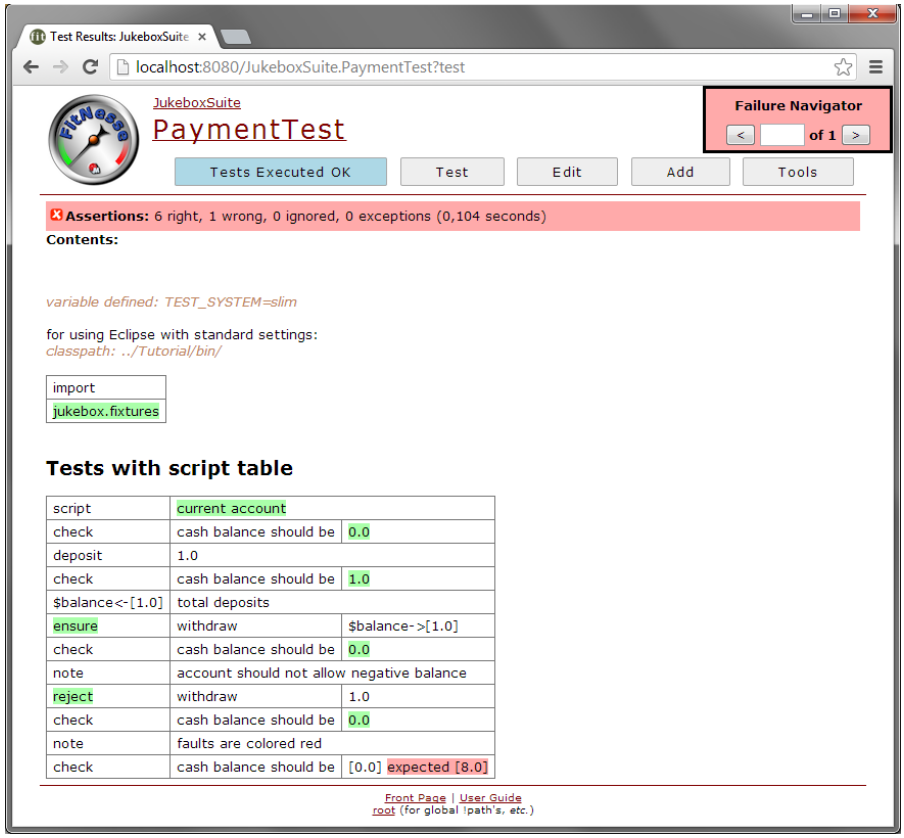


Figure 12: Running the test

FitNesse supports two programming languages for these fixtures: Java and DotNet [65]. Support for other languages can be build in manually; some extensions for other languages exists already (e.g., plug-ins for Ruby, C, and PHP) [66]. In general, fixture code should be as thin as possible, only piping and wiring between the test table and the application code under test.

4.6.2 Detailed FitNesse Info

Test Systems

FitNesse supports two test systems, FIT (Framework for Integrated Test) [33] and SLIM (Simple List Invocation Method) [34]. FitNesse started as a wiki front-end to FIT and, since 2008, FitNesse has added SLIM as alternative to FIT [67].

Test Tables

The SLIM and FIT test system supports several table styles to define your tests in. The most common used table styles are [34, 68]:

- ColumnFixture (FIT) and Decision Table (SLIM), which allows series of inputs and outputs to be defined.
- RowFixture (FIT) and Query Table (SLIM), which allows testing queries that should return an exact set of values.
- ActionFixture (FIT) and Script Table (SLIM), which allows series of events to be performed.
- Comment Table (FIT) and Comment Table (SLIM), which allows to write a tabular comment that is not executed as test.
- Import Table (FIT) and Import Table (SLIM), which allows to add a path to the fixture search path.

Architecture

The FitNesse architecture with both test systems is shown in figure 13. As shown on the left of the picture, test cases are defined in a wiki in FitNesse. Dependent on the specified test system, either the Fit Client or the Slim runners will be used which eventually execute fixtures that perform test API calls to the system under test. The blue blocks in the picture are given and generally not changed and the orange blocks are the places where application specific development needs to be done.

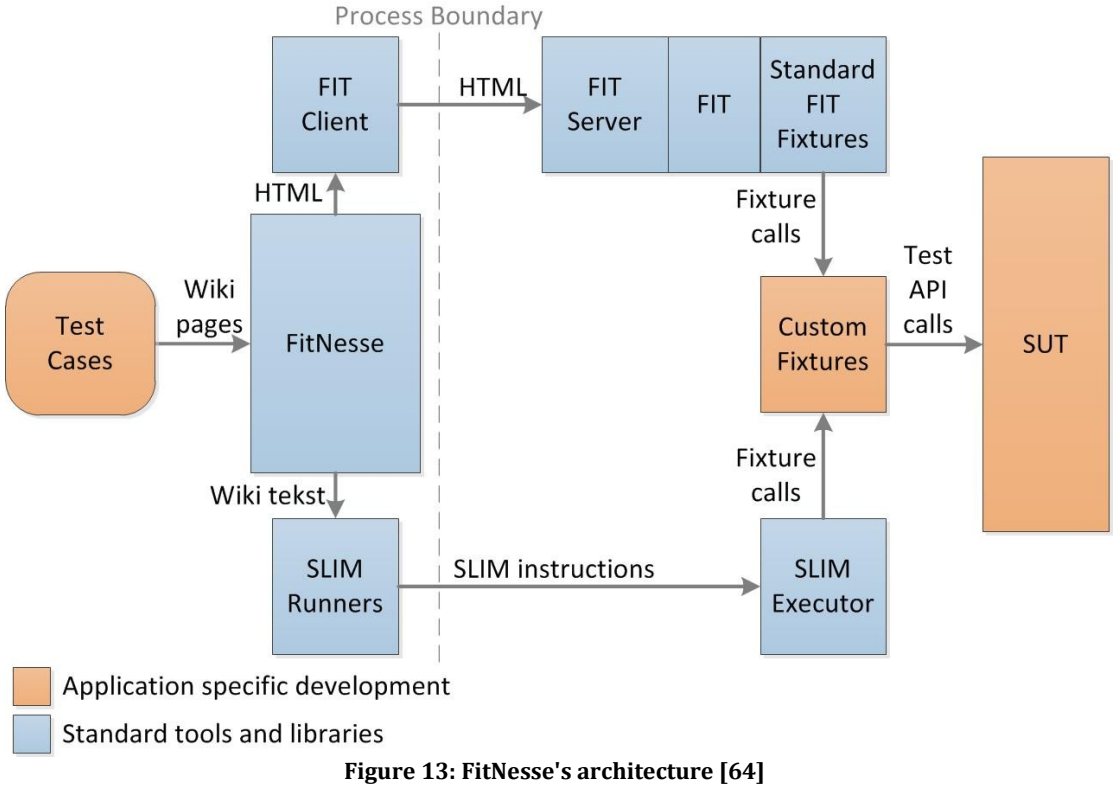


Figure 13: FitNesse's architecture [64]

5 High level Strategy

This section describes the first strategy design, which is a high level design where much details have not been decided. This strategy design was created prior to the case study, so the case study could start the iterative pattern from section 3.3.2 (i.e., observe-identify-develop-test) directly with the test phase: applying the strategy on a real project.

The main goal of the strategy is achieving continuous delivery by introducing test automation and a continuous delivery framework. Continuous delivery consists of several steps: version control, automated build and unit tests, automated acceptance tests, manual acceptance tests and a release. Tools are needed for each of these steps that can be automated, including a continuous delivery framework to manage the continuous delivery process.

The high level strategy below begins with a list of chosen tools for each step and then illustrates how these tools are combined into a continuous delivery workflow. After that, it is described how tests should be created with FitNesse and test-driven development, several test guidelines are given and some specific choices on how to use the tools and environment are made. Finally, this chapter states how the strategy is documented.

5.1 Tools

Version Control Tool: Git (1.8.4)

For version control, Everett uses the version control software Git [26] in combination with Bitbucket [69], which hosts the Git-repository in the cloud.

Automated Build+ Unit Test Tool: Ant (1.0.b3) and JUnit (4.8.2)

Since this project had been using ant scripts to build the software and run the unit tests from JUnit, these ant-scripts had been reused and called by the continuous delivery framework in order to automatically build the software and run unit tests during the continuous delivery cycle.

Automated Acceptance Test Tool: FitNesse (release 20131110) with SLIM test system

FitNesse had been chosen as test automation tool (see section 2.5 on how and why this choice is made). As stated in section 0, FitNesse offers two test systems: SLIM and FIT. The SLIM test system of FitNesse had been chosen for this strategy, since SLIM is newer, easier to use, easier to port to different platforms and more powerful than FIT[34]. See section 5.3 on how to use FitNesse in order to create and run tests.

Integrated Development Environment Tool (IDE): NetBeans 7.3 and 7.4

It is desired to use a good IDE when creating the Java fixtures, because an IDE enhances the productivity of the developer. Since in this project Netflix was been used for small development and NetBeans is an well-known and appropriate IDE, NetBeans has also been used when creating fixtures.

Continuous Delivery Framework: To Be Decided (Jenkins 1.528)

As stated in section 4.2.3, Jenkins and Bamboo had been two possible candidates to act as continuous delivery framework in this project. For the high level design, had been yet undecided which one will be used.

> *In the second iteration, Jenkins has been chosen as continuous delivery framework (see section 6.2.1).*

5.2 Use Continuous Delivery Framework

Use Continuous Delivery Framework

The continuous delivery framework connects all the tools together; it pulls the latest code from Git, runs the automated build and unit tests (and deploys the application) by calling an ant-script, runs the automated acceptance tests of FitNesse and finally shows the results.

> *In the second iteration, it has been chosen to run the continuous delivery framework early in the morning instead of after each commit to save resources (see section 6.2.1).*
>
> *In the fifth iteration, the time-out of the continuous delivery framework had been set to 5 minutes instead of 1 minute, in order to solve a time-out error (see section 6.5.1)*

5.3 Use FitNesse

Use FitNesse with Test-Driven Development

Using test-driven development (TDD) in agile environments offers some benefits. As stated in section 4.4, TDD enhances the test and code quality, makes your code more suitable for testing and sets the focus on testing the actual requirements. As stated in section 4.6, FitNesse works well with TDD. Furthermore, the principles of TDD to test to test as early and often as possible is consistent with the principles of continuous delivery, where tests are performed continuously.

The book from Adzic [62] describes how to combine FitNesse and test-driven development, which I had illustrated in a workflow-diagram (see figure 14). The following workflow is shown visually in figure 14:

- The first step of test-driven development with FitNesse is to let business users and/or developers describe functionality in the FitNesse wiki, and then let them demonstrate this functionality with examples in wiki tables, creating for instance scenario's or a list of input-output values.
- When the wiki has a test case, the functionality can be developed or configured by the developers, then they hook the FitNesse fixture to the system under test, and then the test is run.
    - When the test fails, the developer should go back at an loop; developing or configuring the functionality and/or developing and hooking the fixture, run the test again until it passes.
    - When the test passes and the test case/examples can be refined, start at the first step again and refine the test case. When no refinements can be made, the cycle is completed.

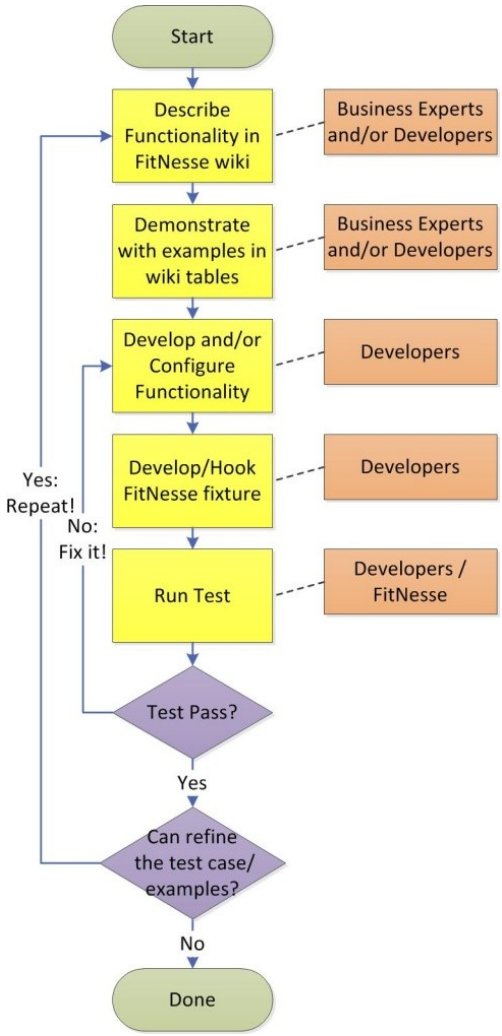


Figure 14: Test-driven development with FitNesse

This workflow is a FitNesse-specific view on test-driven development, first write the test (describe functionality and create wiki tables), then make the test pass (create functionality, hook fixture and run test – if not pass, again), then refactor (not shown in this diagram). It is possible to first develop/hook the fixture and then the functionality, but generally, that would be more difficult if you do not implemented the actions or functions on the SUT that needs to be done or called. When business experts are involved in creating the test cases in the wiki, I had concluded this approach then might be more effective to differ from classic test-driven development; instead of continuously test and develop in steps, it is advised that the business user specifies the complete test in advance so that the business expert can sit down once and isn't constantly involved and interrupted to refine the test case. Then it's the developers task to implement and test the feature in small steps if possible.

Interaction Between Test-Fixtures and IdentityIQ

Since there are multiple ways to perform calls to SailPoint IdentityIQ from outside and none of them are ideal, the pros and cons had been weighed against each other to be able to decide which strategy was the best to use: test via the REST API, test via the console or test via the graphical user interface (GUI).

- REST API: The REST API is made available for customers to use, either from inside a Java application or via HTTP. Therefore, the REST API can be easily called from Java fixtures, offering a couple of methods such as showing/creating/updating identities, check an identity's authorization, aggregate an account, launch workflows [70]. Unfortunately, this set of methods and things you can do or test with these methods is very limited at the time of the project, making the REST API only suitable for some very specific tests and definitely not for acceptance tests.
- Console: The console is a command line utility that makes a live connection to the IdentityIQ database and is mainly used for debugging and troubleshooting. It allows the user to view objects, run tasks, run manager certifications, run workflows, import and export data, and a couple of other functions [71]. The console offers definitely more functionality then the REST-API, but still has functional limitations. Furthermore, the console is harder to use from the Java fixtures then the REST-API and it is a lot slower: the command to start the console is different for each operating system and starting the console initiates the IdentityIQ environment which takes a lot of time (approximately 10-60 seconds depending on the available resources).
- GUI: IdentityIQ is delivered with a web-GUI, which is normally used to interact with IdentityIQ. prior to this strategy, most tests are performed manually in this project, by interacting with the GUI. Although nearly all actions can be performed via the web-GUI, GUI's generally change more often than the functionality-layer below it. These changes could break tests and limit the reuse of these tests on important moments (for instance when a new patch is delivered with new functionalities and a new GUI, the regression tests still needs to work in order to test if the new functionalities did not break anything), where the reusability is a key factor of getting a return on investment on test automation. For this reason, it has been desired to test on a functional level.

With the information above and some experimenting with all possibilities, the choice has been made to use the API when possible, since it is very easy to use from the fixtures, and otherwise use the console, which is much slower but offers more functionalities then the API. Although this won't be the perfect solution, it had been considered the most promising option from the three options above.

In the second iteration, the choice has been reconsidered, since it was not quite usable. An extra option has been found, which replaced this choice: using the private API (see section 6.2.1).
In the third iteration, the SailPoint environment had been initialized and closed before and after the main suite instead of before/after each test (see section 6.3.1).

Collaborate with FitNesse by Storing FitNesse Tests in Version Control (Git)

According to Adzic, here are multiple solutions that allow collaboration between team members using FitNesse [62]:

- Using a single central server and all work on that server. However this is a natural setup, in practice this only works with small teams since it has performing issues when executing multiple tests simultaneously.

- Use a single central server for collaboration but execute tests locally. The central server keeps the updated version of FitNesse tests, which are imported by using a wiki import; developers can then edit test cases remotely and locally. This option needs some discipline in order to have all local updates send back to the central server.
- Store the tests on version control and run local instances of FitNesse. This is a logical option sine most projects use version control for code anyway. The benefit is that both the wiki as the fixtures are in the same repository. Since FitNesse stores the wiki as plain text files in a folder hierarchy and the fixtures are stored as general Java (text) files, version control systems can correctly merge concurrent changes to these files. A downside is that it leaves people like business analysts and customers outside the loop, because they typically do not have tools to access the version control system. This can be solved with an additional "central" test server for people who cannot run FitNesse on their machines.

The most suiting solution for Everett is the latest option. Since Everett uses GIT for their code, it is only natural to use Git for the FitNesse wiki and fixtures as well. Since external version control is used, the advice from Adzic [62] to disable the internal version control of FitNesse is used as well. Keeping it enabled will clutter the version control with zip-files and introduce merge-conflicts on those zip-files. The internal version control can easily be disabled by adding the parameter "-e 0" at the end of the commando that starts FitNesse. As mentioned in the solution, an additional "central" test server must be run for people who cannot run FitNesse on their machines.

5.4 Guidelines

Which tests should be Automated

The guideline from the literature mentioned in section 4.5 had been used as main test automation principle:

> *To estimate the costs and benefits of automating, three questions should be asked [51] (where I've added a fourth question):*
>
> - *How much more time does it take to automate the test instead of manually running it?*
> - *How likely is it that the test dies(e.g., it needs maintenance or is thrown away)? Which events are likely to end it?*
> - *During its lifetime, how likely is this test to find additional bugs (after the first run)? How does this balance with the costs of automation?*
> - *(My fourth question:) Can I re-use (parts) of this test for other tests?*
>
> *Note that there will always be a role for manual testing [51, 61]. First of all, it is the only way to sanity-test the automation itself and secondly, some tests are not worth repeating.*

This guideline has been added to determine which tests should be automated and which not. It helps developers to identify the benefits and costs of automating a test and helps to make a thought-out decision on this.

Since identity solutions projects are short term, the return on investment needs to be sooner than with long-term projects. Therefore it is extra important to focus on reduce the effort to automate a test as much as possible by, for instance, reusing (parts of) tests as much as possible, both in-between tests as in-between different projects. The fact that the projects are short term also gives some extra restrictions in what to test; only test parts with high risks. Furthermore, the developers should beware to not test the third party systems themselves, since it is assumed that those are already tested by the third party themselves and won't deliver the desired return on investment.

The final test suites was been expected to have many tests that test features that are common through several identity solutions projects, such as an aggregation feature, a certification feature, a reporting feature, a form of role management, a form of life cycle management etc. These tests might be reusable for other identity solutions projects as well. It had been expected that the tests also would contain some project or platform specific tests that could have been created quickly in order cover an essential element, rule or task. If such a specific test takes a long time (i.e., multiple hours) to create, it is probably more efficient to test it by hand.

Using Environments for the Continuous Delivery Framework and FitNesse

In the high level strategy, the choice of on which environment(s) the continuous delivery framework, the developers FitNesse instances and the additional central FitNesse instance for business users can or should be run was postponed, since the environment details were not known at that time.

> *In the second iteration it has been decided to run the Jenkins and the FitNesse instance for business users both on the test-environment of the client. The developer may run a test in his local environment or on the development environment as well (see section 6.2.1).*
> *In the fourth iteration, an automatic pull/commit/push-solution had been made in order to keep the separate FitNesse instance and Git both up to date (see section 6.4.1)*

Working with Test Environment Data

In order to have repeatable test suites, a constant initial system state is needed at the beginning of a test run, including appropriate test data. Similar to Enterprise resource planning systems (ERP systems), identity solutions systems are practically impossible to reset to the initial system state: the effort is too high and/or it cannot be done at all [72], which lowers the testability of such systems (i.e. to which degree such systems allow to define and execute an effective testing process).

I had came up with three possible strategies on how to cope with data changes during testing: clean up, leave the changes or reload the initial test data set:

- Clean up: All the changes made during a test needs to be deleted at the end. Sometimes it might not be possible to really delete a change (e.g., creating an identity and the delete command will make it an inactive identity instead of really deleting it).

- Leave the changes: Let the changes stay on the system. This is a simple solution, although you need to make sure that these changes do not stand in the way of a second run of the test or of a run of another test. For instance when creating an identity, random strings can be used to create a new object to avoid "identity already exists"-errors when the test is run multiple times. With this option, the database needs to be cleaned up once in a while to avoid cluttering the database too much.
- Reload the initial test set: After a test or series of tests, reload the initial test set. This will be the most clean solution, since there is always the same solid test environment state when testing. The downside of reloading is that it might be very time-consuming and therefore not practical.

Which option or combination should be used in the strategy was not decided in the high level strategy, but in a group-meeting during the case study since the team members would had a better view of the possibilities and consequences of these options, specifically in identity solutions projects.

> *In the second iteration, it had been chosen to, when possible, clean up the change and otherwise leave the changes be. Once in a while, the test environment must be reset manually. Because of this choice, there is not always a clean data-set. Therefore so no assumptions can be made on the existing data set in the tests (see section 6.2.1).*

Grouping FitNesse Tests in Test Suites

FitNesse tests had been grouped in suites that to group similar tests together. The book from Adzic [62] advices to divide slow and fast tests (to run fast tests on every change and run slow tests every couple of hours) and divide code-oriented and customer-oriented tests (so that customers and business analysts won't get confused by code-oriented tests). This vision was used as starting point for the strategy. In a later iteration, it would be determined if these differences exist our project and, if so, how these should be grouped.

> *In the second iteration, it is decided to not divide slow and fast tests and code-oriented and customer-oriented tests, but group all test suites in one main test suite. Furthermore, from the second iteration forward, the test suite hierarchy was going to be based on functionality (see section 6.2.1).*

Stimulate re-use in Fixtures: Using Inheritance

The FitNesse fixtures were created with the use of Java inheritance to reuse functions as much as possible, both inside the project as for other projects.

I had been making some examples and assumptions on how the reuse with inheritance could take shape in identity solutions projects:

- Inside the project: If features need to be tested that are partly similar, the functions for the similar part can be combined in an own abstraction layer where some functions have a standard body and some are abstract. Sometimes, multiple abstract layers are desirable. If these abstract layers are present, the reusable abstract layer only needs to be implemented once. For all similar tests, only the part that differs needs to be implemented, saving a lot of time.

- Other SailPoint IdentityIQ projects: All the non-application/project-specific parts of the test suite could be reused for other projects, simply by deleting the lowest (specific) classes from the suite above, reusing the abstract classes, and create new classes for the application/project-specific parts of the new project.
- Other identity solutions projects: All identity solutions projects have some part of functionality in common. Reusing the abstract functions (mostly with an empty body, since the body will be mostly IdentityIQ specific) for these common features will give new projects some guidance to start with creating tests for this strategy.

5.5 Documentation

Presentation

A presentation had been made to explain the case study to all team members of the case-study's project; the motivation, goal, method, planning and the most important points of the strategy, referencing to the tutorial for additional information.

Tutorials

Tutorials had been made on Everett's wiki to explain:

- How FitNesse needs to be installed and set-up, what the main functions of FitNesse are, how to use the most common test tables, illustrated with a very simple system under test: decision tables, query tables and script tables.
- How to interact with IdentityIQ from the Java fixtures.
- Advanced FitNesse info.

The goal of these tutorials is to give the developers the knowledge on what FitNesse's capabilities are and, furthermore, how to use it in combination with the IdentityIQ project.

The tutorial parts of FitNesse's installation, setup and main functionalities is based on an existing tutorial by Erik Pragt [73]. That tutorial also uses the SLIM test system with FitNesse and is very clear, although the examples by Pragt were not always complete. Therefore I had been added some more general and project-specific details and the complete code-base to enable team members to experience the functionalities in practice as well. This tutorial had been tested by one of my colleagues, a couple of weeks before the case study, in order to determine if the tutorial was clear and correct. The colleague had found one small error which I corrected (paths are in Mac OS and Linux specified with a forward slash where in windows you could use both forward as backward slashes. The tutorial used backward slashes resulting in an error on the Mac). The colleague also stated that more information is needed on how to use FitNesse to interact with SailPoint IdentityIQ.
With this feedback, the interaction with IdentityIQ had been added to the tutorial. These instructions included a simple Java-example for calling IdentityIQ from both the REST-API as from the console.

The tutorial of FitNesse also includes some advanced FitNesse info, where is the rest of the strategy was defined.

Some small changes had been made during several iterations, making some things a bit more clear.
In the fourth iteration, a tutorial had been added to the wiki on how to automatically push and pull from Git for the running FitNesse instance (see section 6.4.1).
In the fifth iteration, a tutorial had been added to the wiki on how to install and configure Jenkins (see section 6.5.1).

The export of the final tutorial is listed in appendices E, F and G.

6 Case Study: Fine-Tuning the Strategy

As stated in the Research Design in section 1.4 and chapter 3, the strategy has been created and adjusted through iterations. During the case study, the high level strategy was applied, evaluated and if needed, adjusted or supplemented. The strategy was evaluated by holding surveys after each iteration and by holding an interview with all team members after the last iteration.

In order to anonymously discuss opinions and work of the several team members, but keep identities they are labeled with a letter:

- Team member A is from Everett and had been working on the case study project as the architect/scrum master.
- Team member B is from Everett and had been working on the case study project as a developer.
- Team member C is from Everett and had been working on the case study project as a developer.
- Team member D is from the client and had been working on the case study project as an acceptance tester.
- Team member E is from the client and had been working on the case study project as the product owner.

This chapter explains per iteration what had been changed or added to the high level strategy , how the strategy was applied and discusses relevant parts of the evaluation, following the iterative design flow of the case study.

6.1 First Iteration

6.1.1 Adjustments and Additions

The first iteration starts with the high level strategy as described in chapter 5, without any adjustments or additions.

6.1.2 Application

The case study had started with a presentation as introduction, given by me, explaining the strategy to all team members of the case study.

A day after the presentation, team member A installed FitNesse on site and added it to the GIT repository.

After that, team member B started implementing a simple test together with me, in order to investigate how the strategy works in practice. The first test was about testing an aggregation for an application via an CSV-file: running the aggregation and verifying if the aggregation did correlate the amount of identities that were available in the CSV-file. After the first test, further applications of the strategy were put on-hold, since the way of interacting with IdentityIQ via the API and the console is considered quite unusable (see section 6.1.3 for the details). Later on, this test was removed.

6.1.3 Evaluation

In this iteration, no survey was conducted since an important problem had been found, resulting in an on-hold strategy.

When creating the first test, a couple of problems were found when using the console to interact with IdentityIQ:

- The console needed to be started via the Java-runnable method which needs a different call for each operating system. Of course, the Java program can loop over several operating systems and list for each different system what call needs to be made, but that would be an ugly hack.
- The console's feedback/result could only be gathered via file-exports, which need to be read with Java file readers. These readers might have read-locks errors and files might be outdated, so you need to make sure to clean the file afterwards. This cleaning can easily be forgotten and therefore yield wrong results.
- The console needed to be restarted for each sequence of commands. Because starting the console takes a long time, it makes tests far too slow.

Before further adoption of this strategy in this project could have been done, these issues above needed to be solved.

6.2 Second Iteration

6.2.1 Adjustments and Additions

Continuous Delivery Framework

Addition: chose Jenkins (1.528) as framework

As stated in the high level design, a continuous delivery framework is needed in order to achieve continuous delivery. Since a team member on the case study (team member A) had some experience with Jenkins, Jenkins is open source and, next to plug-ins for Ant and Git, even a plug-in for FitNesse was available [74], Jenkins is chosen as the continuous delivery framework.

Use Continuous Delivery Framework

Addition: added details on configurations

In order to not use all resources of the machine where Jenkins and FitNesse are deployed on and not clutter logs, the tests should be run on a time that nothing else runs, for instance around 05:00 to 07:00AM, so the test results are ready when the workday starts. This differs from the continuous delivery cycle in section 4.2.1 which builds and tests after every GIT push, but this choice is needed when there are less resources available.

In the fifth iteration, the time-out of the continuous delivery framework had been set to 5 minutes instead of 1 minute, in order to solve a time-out error (see section 6.5.1)

Interaction Between Test-Fixtures and IdentityIQ

Change: use private API instead of REST-API and console

As stated in the evaluation of the first iteration, the chosen approach for interaction between the fixtures and IdentityIQ was not sufficient. In order to solve this problem, I had held a brainstorm with the team members of Everett which had resulted in a new option: using the private API of the product instead of the REST API and the console. In order to use this, a part of the code of IdentityIQ was decompiled and reconstructed in a java framework.

The created java framework initializes the database-environment of the product, which is normally also done when the console is started. From this environment, functions in the private API can be called directly from Java, including the functions that the console offers. Furthermore, the so called *SailPointContext* can be gathered as well, which can be used to query objects from the database.

This solution is faster than the previous one and uses less workarounds, since now the console functions can be called directly from Java and the environment only needs to be initialized and closed once every test: initialize before the test and close after the test.

In the third iteration, the SailPoint environment had been initialized and closed before and after the main suite instead of before/after each test (see section 6.3.1).

Using Environments for the Continuous Delivery Framework and FitNesse

Addition: run Jenkins and FitNesse instance of client in test-environment and developers may run tests in development environment or local environment.

A project often has multiple environments. In this project, there had been four environments available for the team: development, test, acceptation and production. The production needed to stay clean of tests, since this is the real and critical environment. The acceptation development also needed to stay clean since it is used for manual acceptance tests. This leaves the development and test environment as options for running the continuous delivery framework and FitNesse.

Because the development environment was already been used very intensively, the test environment has been hosting the continuous delivery cycle.

The test environment had also been chosen to run the separate FitNesse instance on, where the non-technical team members can define tests in. Because of his separate FitNesse instance, the non-technical team members don't have to worry about installing Git and FitNesse but simply go to this FitNesse wiki instead. One team member had been made responsible to run this wiki and to push and pull changes on the wiki from and to Git at the end of every day to keep the wiki up to date.

When developing a fixture or functionality, the developer my run a FitNesse instance in his local environment and/or on the development environment. When using the development environment, the developer should only run the

test that he is working on and not the whole suite in order to keep the used resources and interference with others minimal.

In the fourth iteration, an automatic pull/commit/push-solution had been made in order to keep the separate FitNesse instance and Git both up to date (see section 6.4.1)

Working with Test Environment Data

Addition/Choice: clean up when possible, otherwise leave changes and reset the environment once in a while

In order to choose between the test environment data options given in the first iterations (clean up, leave the changes or reload the initial test set), I had held a small group-meeting with the Everett team members on the case study where we together came up with an approach:

- When possible, clean up changes at the end of the test.
- Otherwise, leave the changes be, but make sure these changes don't prohibit a new test run.
- Once in a while reset the test environment and reload the test data from the export. This takes a lot of time (and therefore it cannot be run that often), but cleans the system well.

Since the test environment had also been used for manual tests, the data-import files are updated from time to time and some test-changes cannot be reverted, no assumptions about the existing data set in the tests can be made.

Grouping Tests in Test Suites

Addition: group all tests in one main suite and order sub suites on functionality

As stated in the first iteration, the book from Adzic [62] advices to divide slow and fast tests and code-oriented and customer-oriented tests. However, in the case study, almost all tests would have been slow because of the time it costs to set-up the environment. Furthermore, the difference between code-oriented and customer-oriented tests are not that clear in this project, so at this point, no difference between fast and slow tests and code-oriented and customer-oriented tests had been made. It might be wise to reconsider this choice when necessary.

All suites had been grouped in one main suite called *AllTestSuite*. This way, all tests can be run from that one suite. Before and after this suite, the setup and teardown of the environment is done; When this suite (or any test or suite in this suite) is run, the environment is setup before and tear down after. Furthermore it makes the Jenkins-configuration simple (only pointing at one suite to run all tests).

Tests still need to be grouped in a test suite hierarchy to have all tests ordered in a logical way. The way that had felt the most naturally for this project is to group on the basis of functionality (e.g. aggregation, certification, life cycle manager), so that is chosen as first approach.

6.2.2 Application

I designed a new SailPoint-Interaction-framework on the a demo-environment, as was explained in section 6.2.1.

With this framework, I have been making example tests on the same demo-environment to show how the new interaction with SailPoint IdentityIQ works:

- Two example tests which tests two different aggregations in IdentityIQ. These examples are a good example on how the abstract layers can be used; it had a layer structure of: {Aggregation, AggregationCsv, AggregationCsvApplication}, where the Aggregation layer has reusable methods for all aggregations, the AggregationCsv layer has reusable methods for all csv aggregations and the AggregationCsvApplication has some specific methods and possible some overwrites if it differs from the standard.
- An example test that tests the naming convention when creating an Active-Directory account, specifying how special characters are filtered, how names are shortened and tests which exceptions there are and how the new login-name should be constructed. This test was inspired by a functionality where team member B worked on in the real environment, using BeanShell in IdentityIQ to program the naming convention. The most easiest way to test this functionality is to, instead of using BeanShell, use a Java-class that does all the work, test this Java-class and call the functions of this class from the BeanShell-code in IdentityIQ. For the demo, I extracted the BeanShell code to Java-code and tested this code. When running this test, it found that the apostrophe was not filtered and the special characters were not yet filtered in the prefix of the surname.

On site, Jenkins had been installed on the test environment by team member A. Jenkins pulls from GIT, then calls the build-script which builds the software, runs unit-tests, creates a possible release, and finally runs the FitNesse test suite. When test automation is achieved on site was well, this step is the step that completes the continuous delivery cycle.

At the end of the sprint, during the "Demo & Retrospective", I had given a demonstration that showed the example tests that were made on the small demo-environment. I had shown how they work (with fixtures and Java code), that they use inheritance, that they can be run and that the last suite found an error in the code. This demonstration was given in order to show how FitNesse can be used in such a project and what the added value can be, also for business users. This demo was also used as input for the evaluation that followed.

6.2.3 Evaluation

The first survey was handed out during the "Demo & Retrospective", directly after the demonstration given by me. In total, 5 team members (3 of Everett, 2 of the client) participated at the survey.
the overview of results of this survey are covered in section 7.3.1. Here, only the notable statements are listed:

- Team member A indicated that the acceptance testers have difficulty picturing how to use FitNesse and that redeployment of Java-classes used by IdentityIQ (which are sometimes used instead of BeanShell-code to make testing easier) takes a long time.
- Team member B indicated that when a test is performed, the environment changes, making it possible that a test works on one

environment and not on another. Furthermore, he/she indicated that smart test should be developed, that tests many steps at once.
- Team member C indicated that time needs to be reserved in order to be able to dive in the strategy with high pressure of work on the project already.
- Team member E indicated that the automated tests still needs to be moved and adapted to the test environment and he/she eagerly awaits the next sprints and is convinced that this strategy can deliver an added value.

I have experienced the same problem that one of the team members noticed by myself as well: testing changes the environment and, furthermore, you cannot assume any state of the environment before testing (except maybe that some rule or task exist with a specified name). This makes it harder to write tests that can be performed repeatedly, but is something that is difficult to solve.

I also experienced that team members have difficulty reserving time to apply the strategy due to the high workload. During the evaluation, we decided/agreed that for the following sprint, at least some team members will put some time in creating tests with this strategy. In order to help the acceptant testers, I had decided to help them create their first test-case.

Furthermore I personally had difficulties in finding the right commands in the private API of SailPoint IdentityIQ, however, when developers already work with IdentityIQ and use BeanShell, I believe it will be easier, especially when some example tests are given in forehand. Due to this finding, I filled the generic Java class called *SailPointConsoleCommands.Java* with functions that cover the most-used commands (run a task, aggregation, certification, get the date, delete etc) as an example and basis when more commands needs to be added. These functions can be called from the fixtures when needed.

6.3 Third Iteration

6.3.1 Adjustments and Additions

Interaction Between Test-Fixtures and IdentityIQ

Change: use SetUp and TearDown pages to initialize and close the environment

Instead of initializing and closing the environment before and after each test, it is done before and after the AllTestSuite, using the SetUp and TearDown pages of FitNesse. This makes it much faster to run the AllTestSuite.

6.3.2 Application

I had created two extra simple tests on the demo environment:
- a certification test which tests if a certification can be started.
- a rule test which tests if a rule can be run.

Team member A and I had applied some additional settings on-site in the ant scripts to compile FitNesse tests as well when the software is compiled and build, so the tests can be executed later in Jenkins.

Team member A created a test on-site which tests an export-task of IdentityIQ.

I copied the wiki-page of the naming convention from the demo environment and placed it on-site. After that, team member D added new conditions and exceptions were to it. While team member D was doing this, I helped with the wiki-syntax.

Team member B moved the naming-convention functionality from BeanShell to a Java-class, where the BeanShell code calls the functions in this Java-class in IdentityIQ. Furthermore, team member B created the fixture that stands between the specification of team member D and the Java-class with the naming convention used IdentityIQ. Finally, team member B used the test to develop/change the naming convention to the specification in the wiki.

6.3.3 Evaluation

The second survey was handed out during the "Demo & Retrospective", directly after the demonstration given by me. In total 6 team members (3 of Everett, 3 of the client) participated at the survey. This includes one extra team member from the client compared with the first survey. However, this extra team member is finally excluded from the results, since this member was not involved in the strategy and was not part of the following evaluations.

Similar as the evaluation section of the last iteration (see section 6.2.3), the overview of results of this survey are covered in section 7.3.1. Here, only the notable statements are listed:

- Team member A indicated that the defined test cases in the running FitNesse wiki for business users are not automatically pushed to Git and therefore not in Jenkins as well. A possible solution is introducing a script that does this automatically. Furthermore, he/she indicated that the test effort lies more at the consultant and less at the acceptant testers. More effort of acceptance testers is needed in order to go deeper in the TDD process. Finally he/she also indicated that the next step is to let all test cases run flawlessly, so the Jenkins notifications can be turned on and the impact of the development work becomes visible
- Team member B indicated that Shifting BeanShell to Java code takes more time then programming in BeanShell since it needs a redeploy on every Java-change. Furthermore, after a redeploy, the console needs to be restarted as well.
- Team member D indicated that he/she found the strategy very effective for testing the naming convention

6.4 Fourth Iteration

6.4.1 Adjustments and Additions

Using Environments for the Continuous Delivery Framework and FitNesse

Addition: use automatic pull/commit/push-solution to keep the FitNesse instance and Git up to date

I have been searching for an solution to automatic push and pull to/from the Git repository so that the running FitNesse server and Git stay both up to date.
For this, there are a couple of options:

- use GitHub service hooks [75, 76]
- use Directory Monitor [77, 78]
- use Windows-task with script (or use unix cronjob with the same script if you have a unix environment) [79]
- use Jenkins [80]

I have chosen the third option with a batch script that performs these automatic pull and push-actions for the folder of the FitNesse, since all options are designed to automatically pull and this option is the easiest to customize to push as well. This batch script runs every hour with the windows task scheduler.

Tutorials

Addition: added tutorial on how to automatically push and pull from Git.

The wiki was updated with a part about how to create the batch script and windows task in order to automatically push and pull from and to Git (see appendix F for the complete tutorial).

In the fifth iteration, a tutorial had been added to the wiki on how to install and configure Jenkins (see section 6.5.1).

6.4.2 Application

As stated above, I have build a script and windows-task that automatically pulls and pushed from and to Git. This solution is put in use on-site by team member A.

Furthermore, I created three test on site:

- An aggregation-test that tests if an aggregation is run successfully (reusing large parts from the aggregation-test on the demo-environment).
- A basic certification test that tests if a certification can be run (reusing large parts from the certification-test on the demo environment).
- A basic reporting test that tests if a report can be run (created this test on the demo-environment as well, for demo purposes).

Team member A created two tests:

- A import test, testing the import-task of IdentityIQ.
- A custom test, testing an API class.

Team member C created two tests:

- A provisioning/integration test for two different applications with IdentityIQ (one test per application).

At this point, Team member A, B, C (from Everett) and D (from the client) all created at least one test.

6.4.3 Evaluation

The third survey was handed out during the "Demo & Retrospective", directly after the demonstration given by me. In total 5 team members (3 of Everett, 2 of the client) participated at the survey.

As in the evaluation section of the last iterations (see section 6.2.3 and 6.3.3), the overview of results of this survey are covered in section 7.3.1. Here, only the notable statements are listed:

- Team member A indicated that it is hard to estimate the changes in the tutorial wiki, since these are made gradually and are not clearly communicated at the start of the sprint, that writing a tests sometimes requires almost completely implementing the functionality. In order to solve this, smart entry/exit criteria needs to be made and that there is quite some risk in performing changes in the presentation layer that are not testable.
- Team member B indicated that the things that are tested via the tool are well documented in the tool.

6.5 Fifth Iteration

6.5.1 Adjustments and Additions

Use Continuous Delivery Framework

Change: in the configuration, thetimeout is to 5 minutes instead of 1 minute.

Suddenly, Jenkins started reporting the error: "test report file ....\FitNesse-results.xml was length 0". This error was caused by a time-out during testing. The time-out was first set on 1 minute, and now changed to 5 minutes, which solved the error (running the whole suite now takes 2 minutes).

Tutorials

Addition: added tutorial on how to install and configure Jenkins

The wiki was updated with a part about how to install and configure Jenkins (see appendix G for the complete tutorial).

6.5.2 Application

I have improved the certification test by after running a certification, also check some content and delete the running certification afterwards. This improvement is introduced both on-site and in the demo-environment.

I have added an extra aggregation test on-site to show how the use of abstract classes make it easy to add a new aggregation test.

I have tried to start FitNesse with his own properties file instead of using the property file of the console, but a strange bug occurred which I reported it to the community of the 3rd party software. In the meantime, the property file of the console can be used.

At last, I have restructured some Java-functions for more reuse.

6.5.3 Evaluation

The fourth survey was handed out during the "Demo & Retrospective", directly after the demonstration. In total 5 team members (3 of Everett, 2 of the client) participated at the survey.

As in the evaluation section of the last iterations (see section 6.2.3, 6.3.3 and 6.4.3), the overview of results of this survey are covered in section 7.3.1. Here, only the notable statements are listed:

- Team member A indicated that he/she found no API for handling requests in IdentityIQ, so this is something that is difficult to test outside the GUI. He/she also indicated that a pitfall is that a team needs a dedicated tester to ensure that sufficient test cases are supplied.
- Team member B indicated that he/she found it difficult that multiple users work at the same environment, having a lot of log data that does not belong to his test activities, but to someone else's.
- Team member D indicated that he/she misses a complete test plan and he/she indicated that not much has changed since he/she still does the manual acceptance tests, but that the strategy saved a lot of time and was very valuable for testing the naming convention.

7 Results

The result of the case study is a strategy for test automation and continuous delivery with some example test cases which are defined while using the strategy on the case study, including an evaluation of the strategy from both surveys as interviews. As stated in section 1.2, the strategy exists of tool-selections, tutorials, approaches and guidelines which are documented in a wiki.

This chapter starts with a summary on the final strategy, its application, its evaluation and ends with an analysis of the results and it's evaluation.

7.1 Final Strategy

The final strategy is the high level strategy of chapter 5 with the additions and changes during the case study, described in chapter 6. This chapter gives a short summary of this final strategy.

The tools that we used for continuous delivery are:

- NetBeans (7.3 and 7.4) for developing code
- Git (1.8.4) for version control
- Ant (1.0.b3) for compiling and building (and running unit tests)
- JUnit (4.8.2) for creating unit tests
- FitNesse (release 20131110) for creating (and running) acceptance tests
- Jenkins (1.528) for creating the continuous delivery cycle; automatically pull from Git, compile and build the project (also creating a release), run unit tests, run FitNesse tests.

Test-driven development is included in the strategy; the FitNesse test case in the wiki needs to be written first, then the functionality and then the FitNesse fixture should be created.

A test automation guideline is used which is based on asking yourself a couple of questions to determine which tests needs to be automated and which should better be done manually. This guideline is extra strict for identity solutions projects, since they are very short-term and the return on investment needs to be sooner than with long-term projects.

Some guidelines/choices have been made on how to use FitNesse:

- FitNesse is configured with the SLIM test system.
- Both the wiki and the fixtures are stored in GIT.
- FitNesse is run locally by developers and one main instance is run at the test-environment for business users.
- Changes in Git and on the running wiki are kept up to date by automatically push and pull from and to Git via a script.
- FitNesse tests are grouped in one main suite and ordered in an hierarchy of suites based on functionality.
- FitNesse fixtures use inheritance when possible, in order to be able to reuse as much as possible.

Jenkins is run on the test environment and performs the continuous delivery cycle every day, early in the morning.

In case of IdentityIQ and a lot of identity solutions projects, changes made by a test cannot always be easily reverted to restore the original state of the system, so we need to agree on an alternative approach:

- For the test data, when possible, changes need to be made undone at the end of the test
- When not possible, changes stay on the test environment.
- Once in a while, the test environment is reset manually.

This set-up introduces some complexity, since no assumptions can be made by the test on an existing data set.

Interaction between the fixtures and IdentityIQ take place via an private API, which enables direct communication to the IdentityIQ database and direct use of the console-functions. The environment is initialized once before the main test suite and closed after the main test suite is run.

A tutorial wiki is made that explains how FitNesse needs to be installed and what the main functionalities are, how to interact with IdentityIQ from the Java fixtures, some advanced FitNesse info stating the FitNesse choices above, how to install and use Jenkins and how to configure the automated push/pull solution for the running wiki for stakeholders.

7.2 Application of the Strategy

In total, ten automated tests were created with the designed strategy and Jenkins was set-up to perform the continuous delivery cycle.

Six of ten automated tests were created by team members different from the author. Team-member A, B and C all made at least one test, team member D created a test-case in the FitNesse wiki and team member E did not create any tests.

In creating my tests, I did not follow the test-driven development paradigm, since the functionality was already implemented and I focused on determining whether or not it is possible to test these functionalities and giving some example tests for the other team members.

The continuous delivery cycle was not performed after each commit, but instead every morning at 07:00 AM in order to save resources on the environment and not clutter logs. When desired, the cycle could also be start manually in Jenkins. Jenkins did not send notifications to developers yet, but provides feedback on its web interface.

7.3 Evaluation of the Strategy

7.3.1 Surveys

This chapter discusses some interesting results from the surveys which were held at the end of each iteration (note: except from the first iteration, since no survey was hold then). Parts of these results were discussed already in the

previous sections. In this section, the results are shown as a whole and ordered per question, in order to show some overall opinions and possible trends. The raw (complete) data of the survey results is available in appendix D.

Experience with Strategies and Tools

- The *experience with SailPoint IdentityIQ* was rated fairly stable throughout the iterations, team-members A,B,C, and E all rated him/herself as having much experience and team-member D rated him/herself as average.
- Throughout the iterations, the *experience with FitNesse* and the e*xperience with test-driven development* grew from very few to few experience to few to average experience. See figure 15 for a visual representation of the numbers of individuals (grey) and the average trend (in orange) of the growth of the FitNesse experience.
- The *experience with test automation* grew slightly, but staying at an average experience. See figure 16 for a visual representation of this chart. It is interesting to see that some individuals also show a dip in experience, which is probably due to difficulty guessing where they stand on experience.

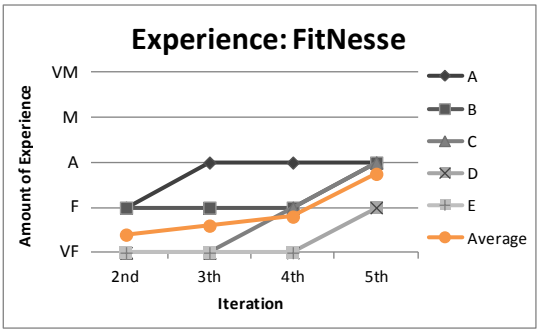


Figure 15: The amount of experience with Sailpoint IdentityIQ. The vertical scale shows abbreviations for: very much, much, average, few and very few.

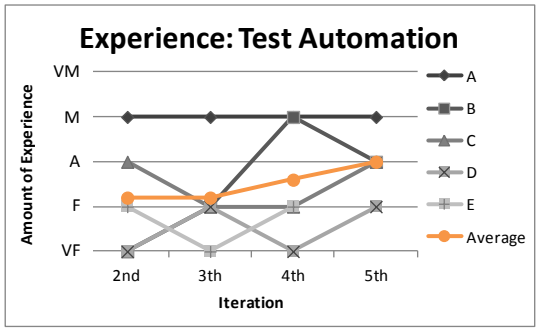


Figure 16: The amount of experience with test automation. The vertical scale shows abbreviations for: very much, much, average, few and very few.

Statements on Strategy and Tutorial

- On average, the team-members agreed on thee following statements:
  - "*Test-driven development lends itself well to start testing early in the development process*".
  - "*FitNesse lends itself well to specify test cases clear and on a central place*".
  - "*FitNesse lends itself well to automate the test cases"*.
  - "*The strategy-tutorial was easy to follow*",
  - "*The explanation on test-driven development is sufficient to be able to apply it in this project*"
  - *The explanation on FitNesse is sufficient to be able to apply it in this project*"
  - "*The explanation on testing IdentityIQ is sufficient to be able to apply it in this project*"
- The statement "*IdentityIQ lends itself well to test outside the web-interface*", was rated on average between neither agreed nor disagreed and agreed. See figure 17 for a visual representation of this chart.

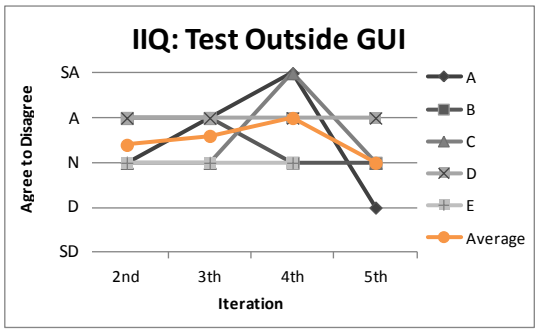


Figure 17: Survey answers on how much they agree with the statement: IdentityIQ lends itself well to test outside the web-interface. The vertical scale shows abbreviations for: strongly agree, agree, neither agree nor disagree, disagree and strongly disagree.

Hours Spend on Testing

- The total amount of *hours on learning/figuring out (parts of) the strategy* was estimated around 3 hours for team member A, 4 hours for team member B and 0 hours for the other team members.
- The total amount of *hours on setting up the test environment and tools* was estimated around 4 hours for team member A, 2 hours for team member B and 0 hours for the other team members.
- The total amount of *hours on defining test cases in FitNesse* was estimated around 2,5 hours for team member A, 1 hour for team member B, 7 hours for team member C, a half hour for team member D and 0 for team member E.
- The total amount of *hours on implementing FitNesse fixtures* was estimated around 3,5 hours for team member A, 7 hours for team member B, 10 hours for team member C and 0 hours for the other team members.
- The total amount of *hours on manual testing* was estimated around 24 hours of team member A (where he/she indicated that he/she did perform a lot of smoke tests after a new deployment and had some trial and error development), 47 hours for team member B (where he/she indicated that he/she tested manually during development), team member C did fill in zeroes and question marks, 18 hours for team member D and 18 hours for team member E.
- The total amount of *hours on other test activities* was estimated around 8 hours for team member A, 9 hours for team member B, team member C did fill in zeroes and question marks,, 0 hours by team member D and 7 hours by team member E.

Problems During Testing

Problems mentioned at the end of the second iteration:

- Team member A indicated that redeployment of Java-classes used by IdentityIQ (which are sometimes used instead of BeanShell-code to make testing easier) takes a long time.

- Team member B indicated that when a test is performed, the environment changes, making it possible that a test works on one environment and not on another.

Problems mentioned at the end of the third iteration:

- Team member B indicated the problem: Shifting BeanShell to Java code takes more time then programming in BeanShell since it needs a redeploy on every Java-change. Furthermore, after a redeploy, the console needs to be restarted as well.
- Team member A indicated that defined test cases in the running FitNesse wiki for business users are not automatically pushed to Git and therefore not in Jenkins as well. A possible solution is introducing a script that does this automatically.

Problems mentioned at the end of the fourth iteration:

- Team member A indicated that writing a tests sometimes requires almost completely implementing the functionality. In order to solve this, smart entry/exit criteria needs to be made and that there is quite some risk in performing changes in the presentation layer that are not testable"

Problems mentioned at the end of the fifth iteration:

- Team member B indicated that he/she found it difficult that multiple users work at the same environment, having a lot of log data that does not belong to his test activities, but to someone else's.
- Team member D indicated that he/she misses a complete test plan.

Statements Comparing with Phase 1

- On the statement "*The documentation on what and how there is tested is improved*", on average, they agreed (3.8 out of 5; lowest was 3; highest was 4).
- On the statement "*Less time is spend on testing*", on average, they disagreed (2.2 out of 5; lowest was 1; highest was 3).
- On the statement "*Bugs are found earlier*", the opinions were scattered. See figure 18 for a visual representation of the numbers of individuals (grey) and the average (in orange). On average, they neither agree nor disagree (3.1 out of 5; lowest was 2; highest was 4).
- On the statement "*More bugs are found*", there might be a slight trend throughout the iterations: from an average rating of 2.8 out of 5 (i.e., neither agree nor disagree) to an average rating of 3.5 out of 5 (i.e., between neither agree nor disagree and agree). See figure 19 for a visual representation of the numbers of individuals (grey) and the average (in orange).
- On the statement "*I have more confidence in the correctness of the project*", there might be a trend throughout the iterations: from an average rating of 2.8 out of 5 (i.e., neither agree nor disagree) to an average rating of 3.8 out of 5 (i.e., agree). See figure 20 for a visual representation of the numbers of individuals (grey) and the average (in orange).

- On the statement "*The strategy has helped with defining clear requirements and test criteria*", on average, On average, they neither agree nor disagree (3.1 out of 5; lowest was 2; highest was 4).

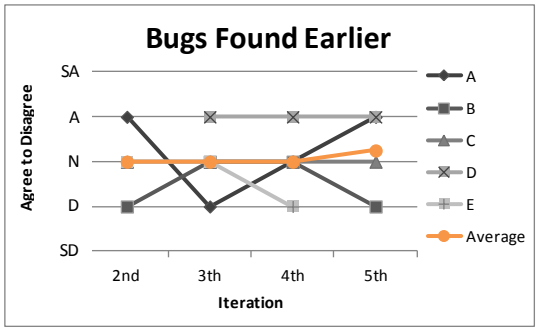


Figure 18: Survey answers on how much they agree with the statement: In comparison with phase 1 of this project bugs are found earlier. The vertical scale shows abbreviations for: strongly agree, agree, neither agree nor disagree, disagree and strongly disagree.

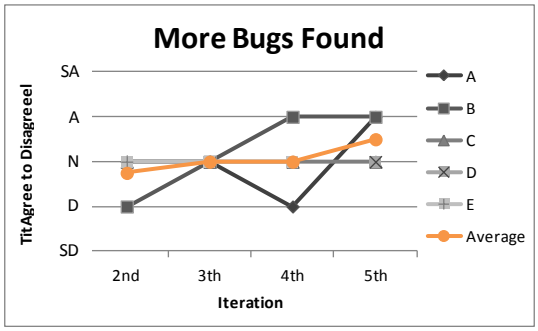


Figure 19: Survey answers on how much they agree with the statement: In comparison with phase 1 of this project more bugs are found. The vertical scale shows abbreviations for: strongly agree, agree, neither agree nor disagree, disagree and strongly disagree.

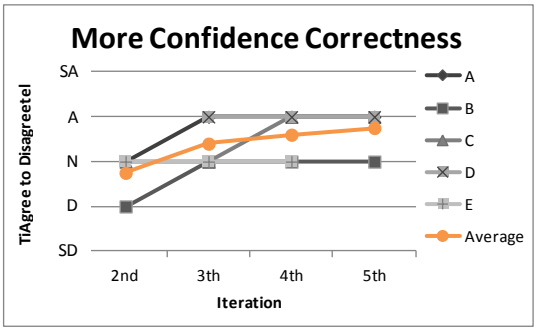


Figure 20: Survey answers on how much they agree with the statement: In comparison with phase 1 of this project I have more confidence in the correctness of the project. The vertical scale shows abbreviations for: strongly agree, agree, neither agree nor disagree, disagree and strongly disagree.

Reuse Test Cases and Fixtures on Other Projects

- Team member A indicated, at the third iteration, that the fixtures are very specific for IdentityIQ and are partly usable for other IIQ projects, but not for other projects; at the fourth iteration, he/she added that these fixtures might be partly reusable for IDM projects as well; at the fifth iteration, he/she indicated that it is quite limited to IIQ, since most fixtures are about IIQ-specific things.
- Team member B indicated, at the end of the second, third and fourth iteration, that many tests are reusable for other similar IIQ projects, but he/she cannot judge for projects outside IIQ. At the end of the fifth iteration, he/she indicated that the tests were quite specific, so the reuse is limited.
- Team member C indicated, at the end of the fourth iteration that the tests and fixtures themselves are not very much reusable, but the experience is; at the fifth iteration he/she added that the tests are to specific for the project of the case study.

- Team member D and E indicated that they cannot give a good estimate on this.

Reuse Strategy Other Projects

- Team member A indicated, through all iterations, that the strategy is definitely reusable for other projects, FitNesse and the strategy are general enough and test-driven development is based on risks and works well on any product. At the end of the fifth iteration, he/she added that it is in particular useful for projects where software is developed. For example at migrations or small projects, a strategy is needed that focuses more on risks then on test automation.
- Team member B indicated, at the end of the second iteration, that it will be reusable for IIQ projects, but he/she cannot say anything about other projects. At the end of the third iteration, he/she noted that it is more generically usable: also at many Access Governance and Identity Management projects. At the end of the fifth iteration, he/she indicated that it is reusable in a reasonable amount, but not everything.
- Team member C indicated, at the end of the fourth iteration, that for IIQ projects, the strategy could be reused and for other projects, the communication with the system under test needs to be re-designed. At the end of the fifth iteration, he/she added that it helps to develop complex connections without continuously having to walk through a complete scenario.
- Team member D and E indicated that they cannot give a good estimate on this.

Additional Comments, Tips, Ideas

Given comments at the end of 2nd iteration:

- Team member B indicated that smart test should be developed, that tests many steps at once.
- Team member C indicated that time needs to be reserved in order to be able to dive in the strategy with high pressure of work on the project already.
- Team member E indicated that he/she eagerly awaits the next sprints and is convinced that this strategy can deliver an added value.

Given comments at the end of 3rd iteration:

- Team member A indicated that the next step is to let all test cases run flawlessly, so the Jenkins notifications can be turned on and the impact of the development work becomes visible.

Given comments at the end of 4th iteration:

- Team member A indicated that he/she is curious about how this strategy can relate to monitoring and unit tests.

Given comments at the end of 5th iteration:

- Team member A indicated that a pitfall is that a team needs a dedicated tester to ensure that sufficient test cases are supplied.

7.3.2 Interviews

Interviews were hold with all 5 team members (3 of Everett and 2 of the client) at the end of the case study. The questions used in these interviews are listed in appendix C. The complete transcript of the interviews are listed in the confidential appendices H to L, which are, by the definition of confidential, non-public. This section offers an objective summary of these interviews.

Costs of the strategy

As costs of the strategy, team member A stated that first of all, all the time that I had spend on the case study (around 30 days) should be considered as a cost, since during these days, the strategy is tailored, adjusted and explained to the team members.

The tutorial did cost team member B around 2 hours to complete, and team member C a couple of hours. Both then know only the basics; just enough to work with it. This indicates that strategy has a short learning curve, as also indicated by team member A. Team member D did not study the tutorial through, but managed to make, together with me, some extra test cases to an existing test and indicated that the tests of the demo were very understandable and readable enough. Team member E added to this that it might have costs hem some time, but he/she did not mind because of the academic value and he/she did not think that the invested time hindered the project.

Team member B indicated that it costs a lot of time to create a test, but the more tests you make, the easier it will become. Somewhat contradictory, team member C indicated that it did not cost him/her more time to automate the test compared with manually testing.

The set-up of Jenkins did cost team member A around half a day; however, he/she had some experience with setting up Jenkins already.

Next to time investments, team member A indicated that cleanup-tasks, after a test had failed, still needs to be done manually, which is a downside.

A downside mentioned by team member D was that he/she wasn't able to apply the strategy alone, since he/she did not completely understand the language or how to formulate tests in the wiki, however he/she indicated that this can be learned and he/she managed with some help.

Benefits of the strategy

One benefit of this strategy all team members mentioned, is that the strategy can be, in the future, very valuable for regression testing and re-running tests, if enough test cases are made.

Team member B and E mentioned that, at the moment for this project, the strategy did not help the acceptation process; for this, more tests needs to be written. Furthermore, team member E has the impression that the real value lies in the development process instead of the acceptation process.

Team member B and D indicated that they were very pleased by the possibility that the client can specify a lot of variants and exceptions with FitNesse, before the developer programs/configures it, making it an easy, structured, more complete and an efficient method. Team member D added to that that he/she then did not have to test every case manually anymore.

Team member C indicated that it could especially help with testing things outside the 3rd party software, implementing with test-driven development in small iterative steps and being able to test this one operation at a time, instead of every time walking through the whole workflow as was needed with manual testing.

Team member A and B indicated that another added value is on the bigger picture as well: the strategy including the communication framework is designed and the involved team members had gathered knowledge of the strategy, continuous delivery and test-driven development.

Team member B, C and D did say that the strategy could help with giving some trust to the client if the client is involved, since it is easy to demonstrate to the client which tests were run and possible to run them again.

Execution of the strategy

Team member A and B indicated that they did invest less in the strategy then was expected in the beginning, partly because the strategy was still under development. Team member C indicated that he/she started late with the strategy. Furthermore, as team member A indicated, we did look more into the possibility to test several aspects of the system and less in what should really be tested at the moment. Furthermore, both team member A and C indicated that we did not yet were on the stage were Jenkins sends email notifications, which should be done in the future.

(Re)use strategy in other projects

All team members of Everett agreed that quite a lot of the strategy can be reused for other projects who also use IdentityIQ as software, not all tests are usable in a new project, since most tests are very project-specific. Team member A and C stated it should be re-used in new IdentityIQ project, where team member B indicated that, for this, a dedicated tester is really needed to watch over the testing strategy and its adoption. Team member A also said that the strategy would have more value if there is an dedicated tester who takes the ownership of the strategy.

For other projects, all team members of Everett agreed the strategy itself can be reused, but the framework on how to communicate with the product needs to be redesigned for each product.
Team member A added some critical notes to that statement, First of all, that it is definitely worth to consider using this strategy for other projects in the area of Identity & Access management, but it is needed to investigate in the amount of work that it takes to integrate that product within the FitNesse fixtures. If this is difficult or takes more time than a week, than the strategy should probably not be used for that project. For completely other projects, it really depends since

other factors may rise, and some parts of the strategy should be reviewed again, so probably this is only beneficial if it is a longer project.

Return on investment

Team member A stated, that for the case study, the strategy didn't outweigh the costs, since the strategy is also designed here and it was the first time. Team member B stated that it did outweigh the costs for the naming convention test specifically and team member A and C mentioned that it probably would for a new IdentityIQ project; and team member C mentioned that he/she would like to add it to the IdentityIQ Accelerator (which is a startup used as basis for each IdentityIQ project).

Furthermore, Team member A expects that the added value of this strategy would be far more clear and easier to achieve at a product company as well and team member B indicated that he/she is not sure if IdentityIQ is the best product for this strategy, since it has a lot of configuration and not much development.

Change in return on investment for different kind of tests and features

On the different added value on different kind of tests, all team members of Everett have their own opinion:

- Team member A indicated that the most value lies in system tests, acceptation tests and integration tests and that unit tests should stay in JUnit or Xunit, since these are integrated in the IDE and are therefore faster to run then FitNesse tests.
- Team member B indicated that he/she sees the most value in specifying very big tests on a high level, from which can be determined if the lower-level steps succeeded or not.
- Team member C did see most value in testing developed software that acts outside of IdentityIQ, so the functionality could be developed in small steps, which can be arranged as a test scenario.

On different added value for different features, all team members also had their own ideas:

- Team member A stated that the most important dependency on the added value for different features is to which extent the feature can be controlled from the outside.
- Team member B stated that the strategy is probably most useful in parts where software is developed and less where software is configured, since when you develop, tests can be build gradually with the deployment.
- Team member C stated that some features of the product is definitely more work to test automatically then others. Especially pieces where a lot of time can sit in between steps (e.g., timeouts and wait for processes), they can be difficult for testing. These tests should be limited to pieces that do not suffer from these time-constraints together with one big manual test.

Train Client

Team member B, C, D and E indicated that the client should be trained more for this strategy, in order to let them create good tests in the wiki, having the client involved and create more trust. According to team member D, if it costs for

instance half a day, this time will be earned back in the long run. Team member E added that it is the question if their company agrees with this, but if the time investment could be done inside the project-scope, he/she would invest in a training. Team member A indicated that, whether or not to train the client depends on the type of client; if the client has business-oriented employees on the project, the conversion from a original test-script to FitNesse should be done by the developers themselves and if the client has technical-oriented employees on the project, they will probably learn it themselves to use this tool. Team member E also indicated that, if the tool is used during the project, he/she would consider keep using the tool afterwards.

Goals

The goal to introduce continuous delivery is considered achieved, however, both team member A and C indicated that we did not yet were on the stage were Jenkins sends email notifications, which should be done in the future. Team member C stated that from earlier experiences with automatic builds with Jenkins, that such a setup is very useful.

The goal to automate tests is in the opinion of team member A achieved, and according to team member B achieved for the parts where tests are written. Team member C found it difficult to estimate.

The goal to spend less money (time) on testing is according to team member A, B and C not yet achieved, since it did cost a lot of time to create the private-API framework, the team members needed to learn the strategy in this project, where sometimes the time in-between creating a test was quite long, resulting in time needed to revisit the workings of FitNesse. Furthermore, there were still quite much manual tests and sometimes a feature is tested both manual as automatic since we did not fully went for the strategy. Team member D indicated that for the test he/she made, it was more effective and costs less time for him/her then testing it by hand. Team member E had the feeling that developers were less occupied with testing. Team member C expects that, when this is applied throughout the company and when everyone is familiar with it, that it would finally save some time.

The goal to have less errors in products does, according to team member A, B and C, completely depends on the tests being written; these tests should cover good scenario's and should be thought out well. Team member A stated it is achieved for the parts that were used for FitNesse, such as the naming convention, since no bugs were found manually on these features. Team member C stated a better quality is achieved for the OVIS-connection. Team member A stated that the manual tests did found some bugs that could also be found with FitNesse and if the test-plan was available earlier, these tests could have been transformed into FitNesse tests. Team member B stated that there would especially be less faults when these tests are used for interim deliveries as well. Team member C stated that from earlier experiences with automatic builds with Jenkins, that such a setup is very useful. Furthermore, he/she states that regression bugs in general show up the most when you work in a large team, which are then made visible when tests are automated. Team member D expects

that this strategy will reduce the amount of faults, since things will be tested on a much earlier stage, causing faults to be found much earlier.

7.4 Analysis of the Strategy and it's Evaluation

FitNesse

In general, FitNesse did its job as test automation framework well. The team members agreed in the survey that FitNesse lends itself well to both automate test cases and specify test cases clear and on a central place. Furthermore, the promises of FitNesse that it is easy to learn and deploy/configure are also supported by the surveys and interviews: in about 3-4 hours, it is possible for developers to use FitNesse in a level that you know how to create test cases and fixtures and FitNesse can be run on-site with a simple download and startup call. For business users, it can be a bit more difficult to learn and use the strategy, as is illustrated by the case study: team member D did not study the tutorial and did manage to add some extra test cases to an existing wiki, but he/she needs some training in order to be able to create a test case from scratch alone.

Test-Driven Development

The combination of test automation and test-driven development is a good practice to start testing early, test in small steps and test during development (which were normally done manually).The theory of test-driven development, the survey-results and the interview-results all support this statement.

Jenkins

Jenkins also delivered the promise of being the framework that connects everything together into a continuous delivery pattern. It can be deployed and configured within a day (team member A did it in half a day, but he/she had done this a couple of times before and I copied it and wrote a tutorial for Jenkins in a couple of hours as well) and gives a good visualization of the status of the build and tests. The plug-ins for Git, Ant and FitNesse did play a factor in this, since they make it quite easy to integrate the software with Jenkins.

Communications

The communication between the fixtures and the system under test was changed from the REST API and Console to using the private API, which was quite an improvement in the time it costs a test to run and more possibilities to interact (especially now that the SailPointContext was also available), but it made it also more complex; the private API did not have Javadoc and sometimes you need to search quite a while to find the right calls and parameters. This struggle is also indicated in some way by the surveys, where the team members on average neither agree nor disagree that IdentityIQ lends itself well to test outside the web-interface; for some features it does, for others it doesn't. Because of this, It might be profitable to dive into possibilities for GUI-testing as well.

Combination of tools and techniques

The combination of tools and techniques works quite well for most of the times, however it did costs some time and adjustments to let it all work together (for instance, by also storing the FitNesse tests in Git, so they are everywhere up-to-date required a separate FitNesse instance for the business users and a Git-auto-

pull/commit/push-solution, which can be considered a hack. Next to that, all software and techniques worked quite well together.

Application

As stated in the application-sections, only 10 test cases were created on site. One cause could be the high pressure of work on the project, as stated by in the survey by team member C and another cause could be that the strategy was still under development, resulting in the team to not fully went for the strategy and still many manual tests are performed.

The guidelines of which tests should be automated was mostly used in order to determine the return on investment of this strategy; in order to give some input to answer these questions in a next project, we focused in the case study mostly on testing different aspects and features of the system and less on these questions before automating a test.

As stated in section 6.2.2, Jenkins is configured to build and test the software every day at 07:00AM instead of after every commit, as the continuous delivery cycle states, in order to save resources at the environment and not clutter the logs.

Return on Investment

The strategy has several benefits; it can be very valuable for regression testing; the client can specify a lot of test cases and exceptions, it can be very useful to help with testing parts outside of the third party software and when the client is involved, it can help giving trust as well.

The strategy also has some costs; time needs to be invested to set up the strategy, and for each different software package, time needs to be invested to set up the communication between the fixtures and the system under test; time needs to be invested in automating the test instead of manually test a feature, which can be sometimes time-consuming and sometimes done quickly; in our case, the clean-up tasks needed to be done manually as well.

In the case study's project, the benefits do not currently outweigh the costs; however for some parts, such as the naming convention, it was valuable, the costs of creating the strategy and framework is not yet earned back with the strategy, but expectations are, that for a new IdentityIQ project, it will be, since the framework is already in place, some example tests are available and three Everett team members already learned how to use the strategy and can share their knowledge in future projects. For a new different project, it depends on the length of the project and the possibilities to integrate the software with FitNesse.

Whether or not the return on investment is different for different kind of tests or different features, I partly agree with team member A that unit tests should stay in JUnit or XUnit, however, when the business user has specific requirements for such a part, I believe it can be valuable to put a unit test in FitNesse, as is illustrated by the test on the naming-convention, which could easily also be specified in a unit test, but with FitNesse, the business user can participate in creating the test and good test criteria and see for themselves that the code

passes the test. Although high level tests can be valuable, I do not agree with team member B that the most value will be there; a lot of scenario's will then not be tested and when a test fails it is hard to determine what is broken. I see the point of team member C that, according to him/her, the most value will lie in testing developed software that acts outside IdentityIQ, since generally the strategy will deliver more value for software developments then for configurations and these parts can now be tested easier, without manually performing scenarios in IdentityIQ, which can be quite a time-saver.

Reuse of strategy

The interviews and surveys both mention that strategy can mostly be reused for other projects, although for projects with different software, the communication with the system under test needs to be redesigned. The test cases will mostly be project-specific, except a few tests that test a standard IdentityIQ feature which can be reused in a new IdentityIQ project.

Goals

The goal to introduce continuous delivery is achieved, with the side-note that no feedback-notifications to the developers were set in Jenkins yet (but that is possible). Instead, the feedback was visible on the Jenkins homepage.

The goal of test automation is achieved, as stated by team member B in the interview, at least for the tests that are written. Since these tests cover several different aspects of the system under test, it is shown that test automation can be applied throughout the software.

The goal of having less costs on testing is not yet achieved because the investment in creating the strategy and learning the strategy were together higher than the benefits at this moment. In a next project, some time savings will probably be achieved since the strategy is known and it is easier to estimate when it is profitable to use. Furthermore, as according to team member B and D in the interviews, applying this strategy with test-driven development helps to let the client specify a lot of variants and exceptions with FitNesse, before the developer programs/configures it. Making it an easy, structured, more complete and an efficient method.

The goal of having less errors was partly achieved, since for a couple of tests (i.e., the naming convention test and the integration tests), the creators identified that they probably had less errors with this strategy since the specifications are known on forehand and tested thoroughly.

Train business users

This raises the question whether or not the business users should be trained in using the FitNesse wiki to create tests. The interviews showed that most of the team members indicate that the client should be trained in creating good tests in the wiki, resulting in a client that is involved in testing and in more trust. When they are involved, the strategy might be more useful in the acceptation process as well as the development process and save test time there as well. While team member A indicated that it depends on the type of client, with business-oriented

clients the tests should be converted to FitNesse by the developers and with technical-oriented clients will learn it themselves.
In my belief, the solution lies in between; when the client is eager to learn and change their methods, it will help to give training and create more involvement and trust, and lowers the barrier between automated and manual acceptance tests, resulting in less duplicated tests. However, when the client does not show interest in the strategy, the training would probably not help.

8 Discussion

This section discusses the strategy, the methodology, the implications of the results and the future research for this research project.

8.1 Strategy

The designed strategy is definitely usable in order to achieve continuous delivery with test automation in identity solutions projects. The strategy can be set up in a couple of hours, although it can cost a significant amount of hours/days to set up and design a way of communicating with the system under test from the java fixtures when the system under test does not offer an easy REST-API. Furthermore, team-members learned how to use FitNesse in just a couple of hours, making it suitable for short term projects.

8.2 Methodology

The case study method was very suitable for a proof of concept, since it tested the strategy in a real-world environment, however, it does not provide any statistically valid results, since the strategy is only applied at one case with a small team. This is the main limitation of the methodology.

Unfortunately, the team had a slow start with applying the strategy, since the communication-framework needed to change in the first iteration already. In the second iteration, I had created demo-tests, but no further tests were made and only in the third and fourth iteration everyone started creating tests. This can have had influences on the results, but nevertheless, the strategy did improve during all iterations into a workable strategy for this project, and the delivered tests show the possibilities of the strategy for Everett.

The main strength of the methodology is that a strategy is not only designed, but it is applied, evaluated and improved in practice as well. With this application and iterative approach, flaws in the strategy were found fast (such as communication framework), which might not have been found without the application.

8.3 Implication of results

For Everett, this case study showed the potential of introducing continuous delivery with test automation in their projects. Furthermore, it introduced test-driven development, FitNesse and some testing-guidelines to the team members of the case study. This knowledge is now in-house and can be used in future projects.

The scientific value of this research project is to show how theoretical concepts and paradigms of continuous delivery and test automation can be applied in practice and are suitable and expectedly profitable for short-term integration projects as well as for standard software development, where these techniques were originally designed for.

8.4 Future research

For future research, the strategy should be implemented on full scale, on a lot of different projects in order to investigate further where the breakeven point of the strategy lies and whether or not the expected benefits and costs will work out.

For projects using different software, it should be investigated whether or not the communication between FitNesse and the system under test can be achieved with a low investment.

9 Conclusion

This chapter provides the final conclusions by answering the research questions, defining to which extent the goals are met and giving the limitations and future work of this research project.

9.1 Goals

There were several goals for this research project, the goals mentioned in section 1.2 are:

- *Introduce continuous delivery, in order to:*
  - *enhance the process of software delivery.*
- *Introduce test automation with FitNesse, in order to:*
  - *lower the costs on testing,*
  - *reduce the amount of faults.*

The first goal to introduce continuous delivery is achieved, with the side-note that no feedback-notifications to the developers were set in Jenkins yet (but that is possible). Instead, the feedback was visible on the Jenkins homepage. From earlier experiences of team member C, automatic builds with Jenkins is a very helpful setup in the process of software delivery.

The goal of introducing test automation is achieved as well. Several automated tests show how tests can be automated throughout the product. Whether or not test automation lowers the costs on testing and reduce the amount of faults is interesting. The investment of designing and tailoring the strategy and learning the strategy by the team members was not yet earned back by the benefits of the project. However, for the naming convention, the test was very effective and saved time in manual testing all cases. For the integrations, it did not cost more time to test it via the strategy instead of by hand, but it did increase the quality. For both the naming convention as the integration, the quality is improved, probably leading to less faults, but in order to show this with significance, the strategy should be applied once again where more tests are written.

9.2 Answer on the Research Questions

In order to answer the main research question of this thesis, this section will start answering the two underlining research questions first, ending with answering the main research question.

9.2.1 First research question

The designed strategy provided an answer to the first underlining research question stated in section 1.3:

1. *What is a good strategy to introduce continuous delivery and test automation with FitNesse?*
   a. *Which guidelines and tools are used in this strategy?*
   b. *How will these guidelines and tools be tailored to system integration projects and to each other?*

The final strategy mentioned in section 7.1 used various tools and techniques, selected on the strategy criteria stated in section 1.2, which will be discussed below.

The tools and guidelines are tailored to system integration projects and identity solutions projects in particular, coping with the short project duration and the data-dependency, by selecting tools and guidelines that satisfied the criteria stated in section 1.2 the best, making the when-to-automate guideline more strict and let the developers cope with a non-specific state of data at their tests. Tools and guidelines were tailored to each other by for instance the FitNesse-specific workflow for using test-driven development and by configuring Jenkins to perform the continuous delivery steps with several tools. This configuration was possible as most tools are designed to also be controlled/called from the outside and there were some Jenkins-plugins as well.

The final strategy introduced continuous delivery and test automation with FitNesse, but whether or not the strategy was good is determined through the the satisfaction of the strategy criteria in section 1.2 (shown below) and the answer on the second research question (see section 9.2.2):

- *Efficacy* - *It must tackle the problem in the problem scope.*
- *Flexibility* - *It must be useable in different projects with different circumstances.*
- *Implementation time* - *It must be easy and fast to install, learn and use.*
- *Cost-effectiveness* - *It must have an early return on investment.*
- *Transferability* - *It must enable the client to keep using and maintaining the tool.*

The strategy satisfies these criteria quite well: the strategy makes it possible to automate tests and achieve continuous delivery; the strategy is mostly reusable in different projects with different circumstances (except the communication with SUT must be redesigned for SUT that aren't IdentityIQ and the test cases), the tools can be installed and configured fast and are easy to learn, the return of investment can probably be matched in new projects and the strategy can be completely transferred to the client.

9.2.2 Second research question

The evaluation of the strategy during the case study provided an answer to the second underlining research question stated in section 1.3:

2. *What is are the costs and benefits of applying the strategy?*
    a. *Which effects has the application of the strategy on a project?*
    b. *Is the strategy an improvement compared to the test- and development strategy used in earlier projects?*
    c. *Where is the break-even point to recoup the effort of applying this strategy?*
        i. *How does this differ in several factors of the projects (e.g. different test types, different features, different projects and different software)?*

d. *To which extent is the strategy applicable for other system integration projects?*

The evaluation showed that the application of this strategy now only showed slight effects of better quality and possibly less faults, but it is expected when this strategy is used throughout the company, that there will be less faults and more trust from the client. Because of these effects, the strategy will be an improvement compared to the old test- and development strategy, where only manual tests are performed, but only if the communication between the fixtures and the system under test is easy to create or at least often reusable, since that took a couple of days in the case study. It is difficult to determine a break-even point for this strategy yet, and the opinions of the team-members differ from which factors influence this; one said it was in particular good for testing software outside IdentityIQ, especially since it did not took him more time to create these tests than manually testing it; features where processes need to wait can be difficult to test and might be better testes partly automatic (where these time-constraints aren't in the scope) and partly manually. The strategy would probably be even more valuable for projects where more code is written instead of performing configurations, taking the whole benefits of test-driven development. The strategy can mostly be reused for other projects as well, except the communication with the system under test for projects with different software.

Generally, this strategy has an added value to the project, with costs of time-investments and benefits of better quality and more trust from the client and possibly on the long run, it can create savings in time.

9.2.3 Main Research Question

The answers on both underlining research questions provided an answer to the main research question stated in section 1.3:

- *How can continuous delivery and test automation with FitNesse be introduced in system integration projects?*

Continuous delivery and test automation with FitNesse can be introduced in system integration projects with the final strategy mentioned in section 7.1.

9.3 Future work

For future research, the strategy should be implemented on full scale, on a lot of different projects in order to investigate further where the breakeven point of the strategy lies and whether or not the expected benefits and costs will work out. For this, Everett probably needs to start with another IdentityIQ project and gradually apply the strategy at other software projects too, when the communication between FitNesse and the system under test can be achieved with a low investment.

References

[1] Everett, "Our Company," 2013. [Online]. Available: http://www.everett.nl/en/company/our-company/.

[2] Georgia State University, "Computer Information Systems GSU Graduate Course Catalog 2013-2014: CIS 8020 - Systems Integration," 2013. [Online]. Available: http://catalog.gsu.edu/graduate20132014/subject/cis/.

[3] Everett, "Identity Solutions," 2013. [Online]. Available: http://www.everett.nl/en/identity-solutions-216/.

[4] S. Drenthen, "A Test Strategy for Continuous Testing in Identity Solution Projects - Research Topics Report," University of Twente, 2013.

[5] Selenium, "Selenium - Web Browser Automation," 2013. [Online]. Available: http://docs.seleniumhq.org/.

[6] Watir, "Watir.com | Web Application Testing in Ruby," 2013. [Online]. Available: http://watir.com/.

[7] FitNesse, "FitNesse FrontPage," 2013. [Online]. Available: http://www.fitnesse.org/.

[8] Atlassian, "Home - GreenPepper," 2013. [Online]. Available: http://www.greenpeppersoftware.com/.

[9] Cucumber, "Cucumber - Making BDD fun," 2013. [Online]. Available: http://cukes.info/.

[10] Robot Framework, "Robot Framework," 2013. [Online]. Available: http://robotframework.org/.

[11] SailPoint, "IdentityIQ," 2013. [Online]. Available: http://www.sailpoint.com/products/identity-iq/.

[12] SailPoint, *SailPoint IdentityIQ Product Brochure,* 2013.

[13] D. Hildebrand and D. Rolls, "System and Method for User Access Risk Scoring". Austin, United States Patent 2008/0288330, 14 May 2008.

[14] Everett, "Identity Technologies," 2013. [Online]. Available: http://www.everett.nl/en/identity-technologies/.

[15] Everett, "Involve - Everett's best practice based agile approach for delivering integration projects.," no. Verson 1.3, 2013.

[16] Everett, "Services," 2013. [Online]. Available: http://www.everett.nl/en/services-219/.

[17] Everett, "Identity Management," 2013. [Online]. Available: http://www.everett.nl/en/identity-solutions-216/identity-management/.

[18] Everett, "Identity & Access Governance," 2013. [Online]. Available: http://www.everett.nl/en/identity-solutions-216/identity--access-governance/.

[19] Basel Committee on Banking Supervision, "Risk Management Principles for Electronic Banking," Bank for International Settlements, 2003.

[20] Everett, "Access Management," 2013. [Online]. Available: http://www.everett.nl/en/identity-solutions-216/access-management/.

[21] Everett, "Identity Federation," 2013. [Online]. Available: http://www.everett.nl/en/identity-solutions-216/identity-federation/.

[22] Everett, "Identity Cloud Solutions," 2013. [Online]. Available: http://www.everett.nl/en/identity-solutions-216/identity-cloud-solutions/.

[23] Everett, "Authentication," 2013. [Online]. Available: http://www.everett.nl/en/identity-solutions-216/authentication/.

[24] E. Hossain, M. A. Babar and H. Y. Paik, "Using scrum in global software development: a systematic literature review," *Fourth IEEE International Conference on Global Software Engineering,* pp. 175-184, 2009.

[25] Atlassian, "Software Development and Collaboration Tools | Atlassian," Atlassian, 2013. [Online]. Available: http://www.atlassian.com/.

[26] Git, "Git - Fast Version Control System," 2013. [Online]. Available: http://gitscm.com/.

[27] The Apache Software Foundation, "Maven - Welcome to Apache Maven," 2013. [Online]. Available: http://maven.apache.org/.

[28] The Apache Software Foundation, "Apache Ant - Welcome," 2013. [Online]. Available: http://ant.apache.org/.

[29] H. Kikkers, B. Nienhuis and E. Rutkens, "Testen van informatiesystemen en het gebruik van (geanonimiseerde) persoonsgegevens," *Compact-Kwartaalblad EDP Auditing 3,* no. 3, pp. 29-36, 2009.

[30] Overheid.nl, "Wet bescherming persoonsgegevens," 6 Juli 2000. [Online]. Available: wetten.overheid.nl/BWBR0011468/geldigheidsdatum_29-04-2013/.

[31] G. v. Blarkom and d. J. Borking, "Beveiliging van persoonsgegevens," Registratiekamer. Achtergrondstudies en Verkenningen 23, 2001.

[32] J. Breeman, "Richtlijn Productiegegevens," Bureau Keteninformatisering Werk & Inkomen, 2002.

[33] FIT, "Wiki: Welcome Visitors," 2013. [Online]. Available: http://fit.c2.com/.

[34] FitNesse, "The SLIM Test System," 2013. [Online]. Available: http://fitnesse.org/FitNesse.UserGuide.SliM.

[35] B. Kaplan and J. A. Maxwell, "Qualitative Research Methods for Evaluating Computer Information Systems," *Evaluating the Organizational Impact of Healthcare Information Systems,* pp. 30-55, 2005.

[36] R. Yin, Case Study Research: Design and Methods. - 4th ed., United States of America: SAGE Publications, Inc., 2009.

[37] SailPoint, "IdentityIQ Lifecycle Manager," [Online]. Available: http://www.sailpoint.com/solutions/products/identityiq/lifecycle-manager.

[38] K. S. Pratt, "Design Patterns for Research Methods: Iterative Field Research," in *AAAI Spring Symposium: Experimental Design for Real-World Systems*, Stanford University, California, USA, 2009.

[39] M. Myers, "Qualitative Research in Information Systems," *Management Information Systems Quarterly Discovery,* no. 21, pp. 241-242, 1997.

[40] W. J. Orlikowski and J. J. Baroudi, "Studying information technology in organizations: Research approaches and assumptions.," *Information systems research,* no. 2(1), pp. 1-28, 1991.

[41] P. Darke, G. Shanks and M. Broadbent, "Successfully completing case study research: combining rigour, relevance and pragmatism.," *Information systems journal,* no. 8(4), pp. 273-289, 1998.

[42] H. Klein and M. Myers, "A set of principles for conducting and evaluating interpretive field studies in information systems," *MIS quarterly,* pp. 67-93, 1999.

[43] J. B. A. Iacono and C. Holtham, "Research methods—A case example of participant observation.," *8th European Conference on Research Methodology for Business and Management Studies: University of Malta, Valletta, Malta, 22-23 June 2009:[proceedings],* p. 178, 2009.

[44] R. Evered and M. R. Louis, "Alternative perspectives in the organizational sciences:“inquiry from the inside” and “inquiry from the outside”," *Academy of Management Review,* no. 6(3), pp. 385-395, 1981.

[45] J. Humble, C. Read and D. North, "The deployment production line," *Proceedings IEE Agile 2006 Conference,* pp. 113-118, 2006.

[46] J. Humble and D. Farley, Continuous delivery: reliable software releases through build, test, and deployment automation., Addison-Wesley Professional, 2010.

[47] A. Spillner and H. Bremenn, "The W-Model Strengthening the Bond Between Development and Test.," in *Proceeding of the STAReast '2002* , Conference, Orlando, Florida, 2002.

[48] S. Mathur and S. & Malik, "Advancements in the V-Model.," *International Journal of Computer Applications,* no. 1(12), pp. 30-35, 2010.

[49] N. Ganesh and S. Thangasamy, "New Agile Testing Modes," *Information Technology Journal,* no. 11.6, pp. 707-712, 2012.

[50] D. Hoffman, "Test automation architectures: planning for test automation.," *Proceedings of the International Quality Week,* pp. 37-45, 1999.

[51] B. Marnick, "When Should a Test Be Automated?," in *Proceedings of The 11th International Software/Internet Quality Week*, San Francisco, 1998.

[52] The Apache Software Foundation, "Apache Subversion," 2013. [Online]. Available: http://subversion.apache.org/.

[53] M. Fowler, "Xunit," 2013. [Online]. Available: http://www.martinfowler.com/bliki/Xunit.html.

[54] Jenkins, "Welcome to Jenkins CI!," 2013. [Online]. Available: http://jenkins-ci.org/.

[55] Atlassian Bamboo, "Continuous Integration & Deployment Software," 2013. [Online]. Available: http://www.atlassian.com/software/bamboo/overview.

[56] B. Satrom, "BDD Primer: Behavior-Driven Development with SpecFlow

and WatiN," *MSDN Magazine,* pp. 50-56, December 2010.

[57] W. Aejmelaeus, "Test-driven development," 2009.

[58] D. Saff and M. D. Ernst, "Reducing wasted development time via continuous testing," *Proceedings of the 14th International Symposium on Software Reliability Engineering,* pp. 281-292, 2003.

[59] D. Saff and M. D. Ernst, "An experimental evaluation of continuous testing during development.," *ACM SIGSOFT Software Engineering Notes,* no. 29.4, pp. 76-85, 2004.

[60] B. Boehm, "Software Engineering," *IEEE Transactions on Computers,* Vols. C-25, no. 12, pp. 1226-1241, 1976.

[61] B. Pettichord, "Seven steps to test automation success," in *STAR West*, San Jose, NV, USA, 1999.

[62] G. Adzic, Test Driven. NET Development with FitNesse, United Kingdom: Neuri Limited, 2008.

[63] M. Sorens, "Acceptance Testing With FitNesse, The Overview," Juli 2013. [Online]. Available: https://www.simple-talk.com/dotnet/.net-tools/acceptance-testing-with-fitnesse,-the-overview/.

[64] FitNesse, "What is FitNesse? - One Minute Description," 2013. [Online]. Available: http://fitnesse.org/FitNesse.UserGuide.OneMinuteDescription.

[65] FitNesse, "Multi Language FitNesse," 2013. [Online]. Available: http://fitnesse.org/FitNesse.UserGuide.MultiLanguageFitNesse .

[66] FitNesse, "Extend FitNesse," 2013. [Online]. Available: http://fitnesse.org/PlugIns.

[67] FitNesse, "FitNesse Release 20081115," 2008. [Online]. Available: http://fitnesse.org/.FrontPage.FitNesseDevelopment.FitNesseRelease20081115.

[68] FitNesse, "Fit Table Styles," 2013. [Online]. Available: http://fitnesse.org/FitNesse.UserGuide.FitTableStyles.

[69] Atlassian, "Free source code hosting for Git and Mercurial by Bitbucket," 2013. [Online]. Available: https://bitbucket.org/.

[70] SailPoint, "SailPoint Compass - REST API Integration," 2013. [Online]. Available: https://community.sailpoint.com/docs/DOC-1642.

[71] SailPoint, "SailPoint Compass - IdentityIQ Console Command Reference," 2013. [Online]. Available: https://community.sailpoint.com/docs/DOC-1631.

[72] S. Wieczorek, A. Stefanescu and I. Schieferdecker, "Test data provision for ERP systems. In , 2008 (pp. 396-403). IEEE.," in *First International Conference on Software Testing, Verification, and Validation*, Lillehammer, Norway, 2008.

[73] E. Pragt, "Getting Started with FitNesse," 2013. [Online]. Available: http://refcardz.dzone.com/refcardz/getting-started-fitnesse.

[74] Jenkins, "FitNesse Plugin," 2013. [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Fitnesse+Plugin.

[75] J. Way, "The Perfect Workflow, with Git, GitHub, and SSH," 2011. [Online]. Available: http://net.tutsplus.com/tutorials/other/the-perfect-workflow-

with-git-github-and-ssh/.

[76] B. Van Damme, "Automatic website publishing with Git, GitHub-Style," 2012. [Online]. Available: http://www.bram.us/2012/05/06/automatic-website-publishing-with-git-github-style/.

[77] DevEnterprise.NET, "Directory Monitor," 2013. [Online]. Available: http://www.deventerprise.net/DirectoryMonitor.

[78] Stack Overflow [forum], "Making git auto-commit," 2009. [Online]. Available: http://stackoverflow.com/questions/420143/making-git-auto-commit.

[79] T. Lee, "TortoiseGIT / GIT Tutorial: Hosting a dedicated server with auto commit periodically on Windows 7 and Windows 8," 2013. [Online]. Available: http://www.thehelper.net/attachments/git-tutorial-auto-commit-pdf.18221/.

[80] Google Discussion Groups [forum], "Looking to auto-update a GIT local repo (on Windows Server) when bare repo on redmine is pushed to," 2011. [Online]. Available: https://groups.google.com/forum/#!msg/git-users/9wVCXrZabCE/r3ip0XyM_ZIJ.

[81] FitNesse, "Executing Tests Outside The User Interface," 2013. [Online]. Available: http://fitnesse.org/FitNesse.UserGuide.ExecutingTestsOutsideTheUserInterface.

Appendix A: Principles for Interpretive Field Research

The table below shows the summary of principles for Interpretive Field Research used in the case study, as given by Klein and Myers [42, p. 72].

Principles
1. The Fundamental Principle of the Hermeneutic Circle This principle suggests that all human understanding is achieved by iterating between considering the interdependent meaning of parts and the whole that they form. This principle of human understanding is fundamental to all the other principles. *Example: Lee's (1994) study of information richness in e-mail communications. It iterates between the separate message fragments of individual e-mail participants as parts and the global context that determines the full meanings of the separate messages to interpret the message exchange as a whole.*
2. The Principle of Contextualization Requires critical reflection of the social and historical background of the research setting, so that the intended audience can see how the current situation under investigation emerged. *Example: After discussing the historical forces that led to Fiat establishing a new assembly plant, Ciborra et al. (1996) show how old Fordist production concepts still had a significant influence despite radical changes in work organization and operations.*
3. The Principle of Interaction Between the Researchers and the Subjects Requires critical reflection on how the research materials (or "data") were socially constructed through the interaction between the researchers and participants. *Example: Trauth (1997) explains how her understanding improved as she became self-conscious and started to question her own assumptions.*
4. The Principle of Abstraction and Generalization Requires relating the idiographic details revealed by the data interpretation through the application of principles one and two to theoretical, general concepts that describe the nature of human understanding and social action. *Example: Monteiro and Hanseth's (1996) findings are discussed in relation to Latour's actor-network theory.*
5. The Principle of Dialogical Reasoning Requires sensitivity to possible contradictions between the theoretical preconceptions guiding the research design and actual findings ("the story which the data tell") with subsequent cycles of revision. *Example: Lee (1991) describes how Nardulli (1978) came to revise his preconceptions of the role of case load pressure as a central concept in the study of criminal courts several times.*
6. The Principle of Multiple Interpretations Requires sensitivity to possible differences in interpretations among the participants as are typically expressed in multiple narratives or stories of the same sequence of events under study. Similar to multiple witness accounts even if all tell it as they saw it. *Example: Levine and Rossmore's (1993) account of the conflicting expectations for the Threshold system in the Bremerton Inc. case.*
7. The Principle of Suspicion Requires sensitivity to possible "biases" and systematic "distortions" in the narratives collected from the participants. *Example: Forester (1992) looks at the facetious figures of speech used by city planning staff to negotiate the problem of data acquisition.*

Table 3: Summary of Principles for Interpretive Field Research [42, p. 72]

Appendix B: Survey During Sprint Retrospective

The responses to this survey will be used to evaluate and improve the testing strategy. It is important to give honest answers to this survey. Additional comments, ideas and tips are always welcome.

Name: _____

Experience with products and methods

1. **How much experience do you have with the following products/methods? choose: very much (VM), much (M), average (A), few (F), very few (VF)**

(Check the relevant box)

	VM	M	A	F	VF
SailPoint IdentityIQ	❑	❑	❑	❑	❑
FitNesse	❑	❑	❑	❑	❑
Test-Driven Development	❑	❑	❑	❑	❑
Test Automating	❑	❑	❑	❑	❑

Additions/Comments/Tips:

_____

_____

Strategy design

2. **Indicate for the following statements if you strongly agree (SA), agree (A), neither agree nor disagree (N), disagree (D), strongly disagree (SD) or not applicable (na)** *(Check the relevant box for each statement)*

	SA	A	N	D	SD	na
Test-driven development lends itself well to start testing early in the development process	❑	❑	❑	❑	❑	❑
FitNesse lends itself well to specify test cases clear and on a central place	❑	❑	❑	❑	❑	❑
FitNesse lends itself well to automate the test cases	❑	❑	❑	❑	❑	❑
IdentityIQ lends itself well to test outside the web-interface	❑	❑	❑	❑	❑	❑
The strategy-tutorial was easy to follow	❑	❑	❑	❑	❑	❑
The explanation on test-driven development is sufficient to be able to apply it in this project	❑	❑	❑	❑	❑	❑
The explanation on FitNesse is sufficient to be able to apply it in this project	❑	❑	❑	❑	❑	❑
The explanation on testing IdentityIQ is sufficient to be able to apply it in this project	❑	❑	❑	❑	❑	❑

Additions/Comments/Tips:

_____

_____

_____

Applying the strategy

3. **How many hours did you spend in the <u>finished sprint</u> on the following activities?** *(Fill in the estimated hours)*
   ____ hours on learning/figuring out (parts of) the strategy
   ____ hours on setting up the test environment and tools
   ____ hours on defining test cases in FitNesse
   ____ hours on implementing FitNesse fixtures (including the associated code/scripts/xml files)
   ____ hours on manual testing
   ____ hours on other test activities: _____

4. **Did you encounter problems during one of the test activities? Hereby I am not referring to errors found with testing, but problems with testing itself.** *(Check the relevant box)*
   ❑ Yes ❑ No ❑ Not applicable
   If so:
   a. **Which problems did you encounter?**

   _____
   _____
   _____

   b. **How restrictive or how bad are these problems?**

   _____
   _____

   c. **Are the problems (partly) solved? If so: how?**

   _____
   _____
   _____

5. **Indicate for the following statements if you strongly agree (SA), agree (A), neither agree nor disagree (N), disagree (D), strongly disagree (SD) or not applicable (na)** *(Check the relevant box for each statement)*

In comparison with phase 1 of this project ...	SA	A	N	D	SD	na
... the documentation on what and how there is tested is improved	❑	❑	❑	❑	❑	❑
... less time is spend on testing	❑	❑	❑	❑	❑	❑
... bugs are found earlier	❑	❑	❑	❑	❑	❑
... more bugs are found	❑	❑	❑	❑	❑	❑
... I have more confidence in the correctness of the project	❑	❑	❑	❑	❑	❑
... the strategy has helped with defining clear requirements and test criteria	❑	❑	❑	❑	❑	❑

Additions/Comments/Tips:

_____
_____
_____

6. **To what extent can (parts of) test-cases and fixtures that you have made be reused in this project and in other projects (both IdentityIQ projects as projects with different software/scope)?**

(Fill in your answer)

_____

_____

_____

_____

_____

_____

7. **To what extent is <u>de general strategy</u> (tools/manner of working) reusable in other projects (both IdentityIQ projects as projects with different software/scope)?**

(Fill in your answer)

_____

_____

_____

_____

_____

_____

Extra space for comments, ideas, tips

8. **Do you have additional comments, tips, ideas, complaints or pitfalls on the strategy?**

(Fill in your answer)

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Appendix C: Interview Questions for the Final Evaluation

Positive en Negative aspects

1. What are in your opinion the positive and negative aspects of the strategy?
    a. Can you give an example?
    b. How important are they?
    c. How much impact do they have?

Costs and Benefits

2. In your opinion, how much did it cost to learn and apply the new strategy? (time, effort, problems, researching, learning)
    a. With the gathered experience, how much will it cost to apply this strategy in a new project?
3. In your opinion, what has the strategy delivered in this project? (test quality, less time due reuse, less time due to automatic rerun, amount of found bugs, confidence in the correctness of the code)?
4. In your opinion, do you think the benefits outweigh the costs in this project?
    a. How do you think this will be in future projects?
    b. Can you point out specific factors or parts that play an important role in the cost-effectiveness?
        i. Using different kind of tests (unit tests, system tests, integration tests, acceptation tests)?
        ii. Testing different parts of the product (certification, provisioning, live cycle manager)?
        iii. Testing with different systems (database, IdentityIQ, a connection-protocol)?
        iv. The amount of experience with the strategy, tools and software of the project?

Goal

5. To which extent does the project achieve its goal (less money to test, less faults in the product, automating tests)?
6. Does test automation show its worth? (test faster due reuse or find a regression bug)?
7. To which extent do you think that the strategy can be reused in other projects?
    a. and the test-cases, fixtures and code?
8. To which extent does the strategy help the client?
    a. Does the strategy help with the acceptation process?
        i. Why?
    b. Does the strategy help with giving confidence in the solution?
        i. Why?
    c. Does the client have the knowledge to use the tool after the project ends?
        i. If so: what are the benefits to the client?
        ii. If not: is it desirable to invest in this?

Appendix D: Raw Data of Survey Results

This appendix lists the raw data of the survey results.

Question 1: How much experience do you have with the following products/methods? (5=very much. 4=much, 3=average, 2=few, 1=very few)

SailPoint IdentityIQ

Iteration / Team member	A	B	C	D	E	Average
2nd	4,0	4,0	4,0	3,0	4,0	3,8
3th	4,0	4,0	4,0	2,0	4,0	3,6
4th	4,0	4,0	4,0	3,0	4,0	3,8
5th	4,0	4,0	4,0	3,0		3,8
Average	4,0	4,0	4,0	2,8	4,0	3,7

FitNesse

Iteration / Team member	A	B	C	D	E	Average
2nd	2,0	2,0	1,0	1,0	1,0	1,4
3th	3,0	2,0	1,0	1,0	1,0	1,6
4th	3,0	2,0	2,0	1,0	1,0	1,8
5th	3,0	3,0	3,0	2,0		2,8
Average	2,8	2,3	1,8	1,3	1,0	1,9

Test-Driven Development

Iteration / Team member	A	B	C	D	E	Average
2nd	2,0	1,0	3,0	1,0	1,0	1,6
3th	3,0	2,0	3,0	1,0	1,0	2,0
4th	3,0	2,0	2,0	1,0	2,0	2,0
5th	3,0	3,0	3,0	2,0		2,8
Average	2,8	2,0	2,8	1,3	1,3	2,1

Test Automating

Iteration / Team member	A	B	C	D	E	Average
2nd	4,0	1,0	3,0	1,0	2,0	2,2
3th	4,0	2,0	2,0	2,0	1,0	2,2
4th	4,0	4,0	2,0	1,0	2,0	2,6
5th	4,0	3,0	3,0	2,0		3,0
Average	4,0	2,5	2,5	1,5	1,7	2,5

Comments

- At the end of the second iteration, team member D indicated that he/she only has experience with IIQ on the compliance side and less on the LCM side.

Question 2: Indicate for the following statements if you strongly agree (SA=5), agree (A=4), neither agree nor disagree (N=3), disagree (D=2), strongly disagree (SD=1) or not applicable (na=empty)

Test-driven development lends itself well to start testing early in the development process

Iteration / Team member	A	B	C	D	E	Average
2nd	5,0	4,0	4,0		4,0	4,3
3th	5,0	4,0	4,0	5,0	3,0	4,2
4th	5,0	4,0	4,0	4,0	4,0	4,2
5th	4,0	4,0	4,0	4,0		4,0
Average	4,8	4,0	4,0	4,3	3,7	4,2

FitNesse lends itself well to specify test cases clear and on a central place

Iteration / Team member	A	B	C	D	E	Average
2nd	4,0	3,0		4,0	3,0	3,5
3th	4,0	4,0		4,0	4,0	4,0
4th	3,0	4,0	4,0	4,0	4,0	3,8
5th	4,0	4,0	4,0	4,0		4,0
Average	3,8	3,8	4,0	4,0	3,7	3,8

FitNesse lends itself well to automate the test cases

Iteration / Team member	A	B	C	D	E	Average
2nd	4,0	3,0		4,0	3,0	3,5
3th	5,0	4,0		5,0	4,0	4,5
4th	4,0	4,0	4,0	4,0	3,0	3,8
5th	4,0	4,0	4,0	4,0		4,0
Average	4,3	3,8	4,0	4,3	3,3	3,8

IdentityIQ lends itself well to test outside the web-interface

Iteration / Team member	A	B	C	D	E	Average
2nd	3,0	4,0	3,0	4,0	3,0	3,4
3th	4,0	4,0	3,0	4,0	3,0	3,6
4th	5,0	3,0	5,0	4,0	3,0	4,0
5th	2,0	3,0	3,0	4,0		3,0
Average	3,5	3,5	3,5	4,0	3,0	3,5

The strategy-tutorial was easy to follow

Iteration / Team member	A	B	C	D	E	Average
2nd	4,0		4,0		4,0	4,0
3th	4,0		4,0		3,0	3,7
4th	4,0		5,0		4,0	4,3
5th			4,0			4,0
Average	4,0		4,3		3,7	4,0

The explanation on test-driven development is sufficient to be able to apply it in this project

Iteration / Team member	A	B	C	D	E	Average
2nd	3,0	4,0	3,0		4,0	3,5
3th	4,0	3,0	3,0		4,0	3,5
4th	4,0		4,0	4,0	4,0	4,0
5th	4,0		4,0			4,0
Average	3,8	3,5	3,5	4,0	4,0	3,8

The explanation on FitNesse is sufficient to be able to apply it in this project

Iteration / Team member	A	B	C	D	E	Average
2nd	4,0	3,0	3,0		4,0	3,5
3th	4,0	3,0	3,0		4,0	3,5
4th	4,0	4,0	4,0	4,0	4,0	4,0
5th	5,0		4,0	4,0		4,3
Average	4,3	3,3	3,5	4,0	4,0	3,8

The explanation on testing IdentityIQ is sufficient to be able to apply it in this project

Iteration / Team member	A	B	C	D	E	Average
2nd	4,0	3,0	4,0		4,0	3,8
3th	4,0	4,0	4,0		4,0	4,0
4th	4,0	4,0	4,0	4,0	4,0	4,0
5th	4,0			4,0		4,0
Average	4,0	3,7	4,0	4,0	4,0	3,9

Comments

- Team member A indicated, at the end of the second iteration, that the acceptance testers have difficulty picturing how to use FitNesse. At the end of the fourth iteration, he/she indicated that it is hard to estimate the changes in the tutorial wiki, since these are made gradually and are not clearly communicated at the start of the sprint At the end of the fifth iteration, he/she indicated that he/she found no API for

handling requests in IdentityIQ, so this is something that is difficult to test outside the GUI.

- Team member B indicated, at the end of the second iteration, that he/she did not had time yet to study the tutorial (and did fill in not applicable). At the end of the fourth iteration, he/she did indicate that she still did not follow the tutorials, but that the explanations were clear.

Question 3: How many hours did you spend in the finished sprint on the following activities?

Learning/figuring out (parts of) the strategy

Iteration / Team member	A	B	C	D	E
2nd	1	0	0	0	0
3th	2	0	0	0	0
4th	0	2	0	0	0
5th	0	2	0	0	.
Sum	3	4	0	0	0

Setting up the test environment and tools

Iteration / Team member	A	B	C	D	E
2nd	2	0	0	0	0
3th	2	2	0	0	0
4th	0	0	0	0	0
5th	0	0	0	0	.
Sum	4	2	0	0	0

Defining test cases in FitNesse

Iteration / Team member	A	B	C	D	E
2nd	0	0	0	0	0
3th	1	1	0	0,5	0
4th	1	0	6	0	0
5th	0,5	0	1	0	.
Sum	2,5	1	7	0,5	0

Implementing FitNesse fixtures (including the associated code/scripts/xml files)

Iteration / Team member	A	B	C	D	E
2nd	0	0	0	0	0
3th	1	7	0	0	0
4th	2	0	6	0	0
5th	0,5	0	4	0	.
Sum	3,5	7	10	0	0

Manual testing

Iteration / Team member	A	B	C	D	E
2nd	2	2	?	8	15
3th	4	5	?	0	0
4th	2	16	0	0	4
5th	16	24	?	10	.
Sum	24	47	0	18	19

Other test activities

Iteration / Team member	A	B	C	D	E
2nd	0	0	?	0	5
3th	0	1	?	0	2
4th	0	8	0	0	0
5th	8	0	?	0	.
Sum	8	9	0	0	7

Comments

- Team member A indicated that he/she did do quite much trial and error development in the fourth iteration (he/she had that iteration 2 hours of manual tests) and indicated that, in the fifth iteration, he/she did perform smoke tests after a new deployment (he/she had that iteration 16 hours of manual tests)
- Team member B indicated that he/she tested manually during development in the second iteration (he/she had that iteration 2 hours of manual tests).

Question 4: Did you encounter problems during one of the test activities? Hereby I am not referring to errors found with testing, but problems with testing itself.

Problems mentioned at the end of the second iteration:

- Team member A indicated that redeployment of Java-classes used by IdentityIQ (which are sometimes used instead of BeanShell-code to make testing easier) takes a long time
- Team member B indicated that when a test is performed, the environment changes, making it possible that a test works on one environment and not on another.

Problems mentioned at the end of the third iteration:

- Team member B also indicated this problem: Shifting BeanShell to Java code takes more time then programming in BeanShell since it needs a redeploy on every Java-change. Furthermore, after a redeploy, the console needs to be restarted as well.
- Team member A indicated that defined test cases in the running FitNesse wiki for business users are not automatically pushed to Git and therefore not in Jenkins as well. A possible solution is introducing a script that does this automatically

Problems mentioned at the end of the fourth iteration:

- Team member A indicated that writing a tests sometimes requires almost completely implementing the functionality. In order to solve this, smart entry/exit criteria needs to be made and that there is quite some risk in performing changes in the presentation layer that are not testable

Problems mentioned at the end of the fifth iteration:

- Team member B indicated that he/she found it difficult that multiple users work at the same environment, having a lot of log data that does not belong to his test activities, but to someone else's.
- Team member D indicated that he/she misses a complete test plan.

Question 5: Indicate for the following statements if you strongly agree (SA=5), agree (A=4), neither agree nor disagree (N=3), disagree (D=2), strongly disagree (SD=1) or not applicable (na=empty): In comparison with phase 1 of this project.

The documentation on what and how there is tested is improved

Iteration / Team member	A	B	C	D	E	Average
2nd	3,0	2,0			4,0	3,0
3th	3,0	4,0	3,0	2,0	3,0	3,0
4th	4,0	4,0	3,0	3,0	3,0	3,4
5th	4,0	3,0	3,0			3,3
Average	3,5	3,3	3,0	2,5	3,3	3,2

Less time is spend on testing

Iteration / Team member	A	B	C	D	E	Average
2nd	2,0	2,0	3,0		3,0	2,5
3th	1,0	1,0	3,0		3,0	2,0
4th	2,0	2,0	2,0		3,0	2,3
5th	2,0	2,0	2,0			2,0
Average	1,8	1,8	2,5		3,0	2,2

Bugs are found earlier

Iteration / Team member	A	B	C	D	E	Average
2nd	4,0	2,0	3,0		3,0	3,0
3th	2,0	3,0	3,0	4,0	3,0	3,0
4th	3,0	3,0	3,0	4,0	2,0	3,0
5th	4,0	2,0	3,0	4,0		3,3
Average	3,3	2,5	3,0	4,0	2,7	3,1

More bugs are found

Iteration / Team member	A	B	C	D	E	Average
2nd	3,0	2,0	3,0		3,0	2,8
3th	3,0	3,0	3,0		3,0	3,0
4th	2,0	4,0	3,0		3,0	3,0
5th	4,0	4,0	3,0	3,0		3,5
Average	3,0	3,3	3,0	3,0	3,0	3,3

I have more confidence in the correctness of the project

Iteration / Team member	A	B	C	D	E	Average
2nd	3,0	2,0	3,0		3,0	2,8
3th	4,0	3,0	3,0	4,0	3,0	3,4
4th	4,0	3,0	4,0	4,0	3,0	3,6
5th	4,0	3,0	4,0	4,0		3,8
Average	3,8	2,8	3,5	4,0	3,0	2,8

The strategy has helped with defining clear requirements and test criteria

Iteration / Team member	A	B	C	D	E	Average
2nd	3,0	2,0			3,0	2,7
3th	3,0	4,0	3,0	4,0	3,0	3,4
4th	4,0	3,0			3,0	3,3
5th	3,0	3,0	3,0	3,0		3,0
Average	3,3	3,0	3,0	3,5	3,0	3,1

Comments

- Team member A indicated, at the end of the third iteration, that the test effort lies more at the consultant and less at the acceptant testers. More effort of acceptance testers is needed in order to go deeper in the TDD process.
- Team member E indicated, at the end of the second iteration, that the automated tests still needs to be moved and adapted to the test environment.
- Team member D indicated, at the end of the third iteration, that he/she found the strategy very effective for testing the naming convention. At the end of the fourth iteration, he/she indicated that the things that are tested via the tool are well documented in the tool. At the end of the fifth iteration, he/she indicated that not much has changed since he/she still does the manual acceptance tests, but that the strategy saved a lot of time and was very valuable for testing the naming convention.

Question 6: To what extent can (parts of) test-cases and fixtures that you have made be reused in this project and in other projects (both IdentityIQ projects as projects with different software/scope)?

- Team member A indicated, at the third iteration, that the fixtures are very specific for IdentityIQ and are partly usable for other IIQ projects, but not for other projects; at the fourth iteration, he/she added that these fixtures might be partly reusable for IDM projects as well; at the fifth iteration, he/she indicated that it is quite limited to IIQ, since most fixtures are about IIQ-specific things.
- Team member B indicated, at the end of the second, third and fourth iteration, that many tests are reusable for other similar IIQ projects, but he/she cannot judge for projects outside IIQ. At the end of the fifth iteration, he/she indicated that the tests were quite specific, so the reuse is limited.
- Team member C indicated, at the end of the fourth iteration that the tests and fixtures themselves are not very much reusable, but the experience is; at the fifth iteration he/she added that the tests are to specific for the project of the case study.
- Team member D and E indicated that they cannot give a good estimate on this.

Question 7: To what extent is de general strategy (tools/manner of working) reusable in other projects (both IdentityIQ projects as projects with different software/scope)?

- Team member A indicated, through all iterations, that the strategy is definitely reusable for other projects, FitNesse and the strategy are general enough and test-driven development is based on risks and works well on any product. At the end of the fifth iteration, he/she added that it is in particular useful for projects where software is developed. For example at migrations or small projects, a strategy is needed that focuses more on risks then on test automation.
- Team member B indicated, at the end of the second iteration, that it will be reusable for IIQ projects, but he/she cannot say anything about other projects. At the end of the third iteration, he/she noted that it is more generically usable: also at many Access Governance and Identity Management projects. At the end of the fifth iteration, he/she indicated that it is reusable in a reasonable amount, but not everything.
- Team member C indicated, at the end of the fourth iteration, that for IIQ projects, the strategy could be reused and for other projects, the communication with the system under test needs to be re-designed. At the end of the fifth iteration, he/she added that it helps to develop complex connections without continuously having to walk through a complete scenario
- Team member D and E indicated that they cannot give a good estimate on this.

Question 8: Do you have additional comments, tips, ideas, complaints or pitfalls on the strategy?

Given answers at the end of 2nd iteration:

- Team member B indicated that smart test should be developed, that tests many steps at once
- Team member C indicated that time needs to be reserved in order to be able to dive in the strategy with high pressure of work on the project already
- Team member E indicated that he/she eagerly awaits the next sprints and is convinced that this strategy can deliver an added value.

Given answers at the end of 3rd iteration:

- Team member A indicated that the next step is to let all test cases run flawlessly, so the Jenkins notifications can be turned on and the impact of the development work becomes visible.

Given answers at the end of 4th iteration:

- Team member A indicated that he/she is curious about how this strategy can relate to monitoring and unit tests.

Given answers at the end of 5th iteration:

- Team member A indicated that a pitfall is that a team needs a dedicated tester to ensure that sufficient test cases are supplied.

Appendix E: FitNesse Tutorial

This appendix shows the FitNesse tutorial that can be followed in order to use the strategy. Text in black are steps to be taken and steps in gray give extra information.

Tutorial - FitNesse: Installing, Strategy, Usage etc

0. Introduction

The content below follows a nice tutorial (see http://refcardz.dzone.com/refcardz/getting-started-FitNesse) , but this page will give some more detailed or complete code and instructions where the tutorial misses out.
Please follow the whole tutorial, step 1 to 7.

1. FitNesse Setup

Install an IDE to your choice

- Choose and install an IDE according to your preferences, e.g.:
  - Install NetBeans from: https://netbeans.org/downloads/ (I used the all-languages version 7.3.1)

Install Java JDK

- If the Java JDK is not installed, download the Java JDK from http://www.oracle.com/technetwork/Java/Javase/downloads/ and install it.
  - Note: in Windows, you need to add to the path of the bin-folder of the JDK (like: [...]\Java\JDK_xxx\bin) to the PATH environment variable of your computer

Download FitNesse

- If you want to run FitNesse by itself, You could download the FitNesse-standalone.jar from: http://FitNesse.org/FitNesseDownload. It contains all dependencies.
  - Note: If you use Maven, you could find it in the FitNesse Maven Central Repository or download the FitNesse.jar from FitNesse's latest stable build.

Create Tutorial Project

- Create a tutorial project
  - in NetBeans:
    - File - New Project
    - Java project - Java Application - Next
    - Project Name: "Tutorial", Project Location "yourchoice"
    - unselect create main class
    - Finish

Add FitNesse-library to project

- Move FitNesse-standalone.jar to the Tutorial-folder of your project (or when used for your project: to the project-git directory)
- Add the FitNesse-standalone.jar to the Library of the project
    - in NetBeans:
        - Right-click on Libraries and click "add JAR/Folder", browse to .../Tutorial/FitNesse-standalone.jar and click on "open".

Start FitNesse

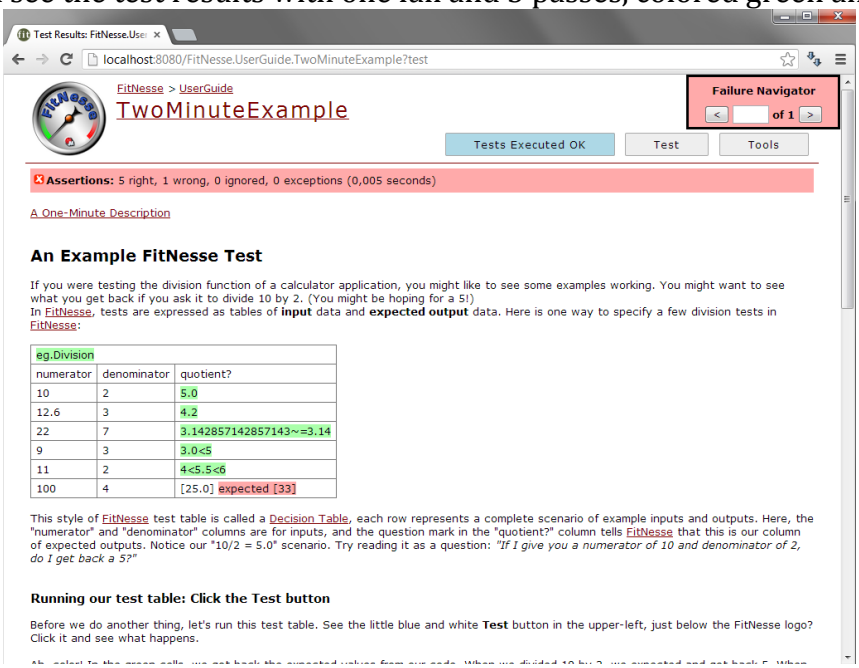
- go to the Tutorial directory.
- Open a command prompt /shell in the Tutorial-folder and type:
    - Java -jar FitNesse-standalone.jar -p 9090 -e 0
      This will extract itself (creating a directory named FitNesseRoot with all its content) and try to run itself.
        - The -p 9090 runs FitNesse on the port 9090 instead of the standard 8080 (because SailPoint uses 8080)
        - The "-e 0" argument disables the built-in versioning system of FitNesse, since we use GIT and leaving it on will clutter the GIT repository.
        - For more possible arguments, see: http://FitNesse.org/FitNesse.UserGuide.CommandLineArguments
        - Note: The command above assumes you moved fitnesee-standalone.jar to the project folder that is in git-sourcecontrol, it starts FitNesse from the location of the jar-file, placing the FitNesse-wiki-pages at this location in a subdirectory called FitNesseRoot. If you don't start the FitNesse-jar-file from the project/git directory but from another folder, you should start FitNesse with an extra argument: -d <pathToProjectDirectory>, in order to start FitNesse in the shared project-directory and share the wiki-pages.
- Let it run (so don't close/kill the process)

FitNesse and GIT

- The content of the FitNesse wiki is stored in the FitNesseRoot-directory that is created when you start FitNesse. If you performed the step above in the right way, the FitNesseRoot-directory is in the project folder of the project you want to test, that is under version control (in our case: git).
- The wiki-pages of FitNesse are stored in this FitNesseRoot-directory. The name of a wiki page is stored as a directories and subpages are subdirectories. Each directory has the page content stored in a txt file and the page configurations stored in an xml file, making them very suitable to submit to GIT (since text-based files are easy to merge)
- The following FitNesse-folders should be in the git-ignore file:
    - FitNesseRoot/files
    - FitNesseRoot/Errorlog
    - FitNesseRoot/RecentChanges

Try FitNesse

- Try the pre-defined FitNesse example:
- Go to localhost:9090/FitNesse.UserGuide.TwoMinuteExample
- Press test at the right corner and it should look like the picture below (click to enlarge).
- You see the test results with one fail and 5 passes, colored green and red:



2. Download Tutorial files, run FitNesse, copy system under test and create a test suite

Download The Tutorial files

- Now you have a working FitNesse server and you could continue with the tutorial.
  - Download Tutorial-Files.zip[2] in which you will find the needed files for this tutorial.
  - (If desired, The final result is also available to use as reference):
    - Tutorial-final-results-NetBeans.zip[1]

Copy the System Under Test from my zip-file

- Go to the Tutorial-Files.zip (downloaded above) and copy the src-folder to your Tutorial-folder.
- The src-folder contains the system under test, which is a very simplified Jukebox, which could calculate how much credits you will get for a certain payment, which has a list of songs, a song could be added and you could find a song from an artist name. From the song, you could get the title, the artist and the duration.

Run FitNesse

- Start the FitNesse server as given in the set-up (run "Java -jar FitNesse-standalone.jar -p 9090 -e 0") and leave it run.
- Go to localhost:9090 , you should see the FrontPage of FitNesse with a Welcome message.

[2] These files can be requested by sending an email to: s.drenthen@alumnus.utwente.nl

Create a Test Suite

- FitNesse has the concept of suites and tests. Suites are sets of tests, which is a way to organize the tests. As an additional benefit, when executing a suite, all tests within the suite are executed.
- Click on Edit in the menu at the top right.
- Insert under the existing text the following and click on save afterwards:

```
!1 The tutorial-pages
JukeboxSuite
```

- This will create a header and a link to a new non-existing link. Note that FitNesse only creates links when the text is written in CamelCase.
- Click on the question mark next to JukeboxSuite and click on save. This creates an empty suite at http://localhost:9090/JukeboxSuite.
  Note: FitNesse marks a page as a Suite automatically when it starts or ends with Suite. A Wiki page can also manually be set as an Suite/Test/Static in the page properties by clicking 'Tools-Properties' when you are at that page.

3. Test with a Decision Table

Creating a FitNesse Test based on a Decision Table

- When you are at http://localhost:9090/JukeboxSuite, click on Add - Test page at the menu at the top right.
- We will create a test for the feature: "calculate how much credits you will get for a certain payment". To test this, we will give several payment input and want to check if the given output of the SUT is the same as our definition. We test this with a decision table (see http://www.FitNesse.org/FitNesse.UserGuide.SliM.DecisionTable). A Decision table test supplies inputs checks if the given outputs match.
- Give the test-page the name: PaymentTest (Note that the page name in FitNesse has to written in CamelCase)
- And the help-text: Testing the payment feature
- Insert under the existing text the following:

```
!2 Tests with Decision table
The First four should pass, last two should fail
!|decision:credits for payment|
| payment      | credits?      |
| .25          | 1             |
| 1            | 4             |
| 5            | 20            |
| 10           | 40            |
| 5            | 21            |
| 10           | 45            |
```

- In this case, our test checks if for every .25 of payment, one credit is received (thus for 1 payment, 4 credits and so on). Note that the output value is specified by a question mark and the first four should pass and last two rows of this test should fail (because they give 1 and 5 credits to much).
- Click on Save and then click on the link to the newly created test.
- When you execute this test by clicking on 'Test' in the menu, your test will fail with an exception: Could not find fixture: DecisionCreditsForPayment.

To get the test to work, we need to do two more things: write the Fixture and configure FitNesse correctly.

Creating a Fixture

- The Fixture will be the layer between the production code (the Subject Under Test) and the FitNesse test pages. There are multiple types of Fixtures, and to support the Test above, a Decision Table Fixture is needed.
- Consider line 11-13 from src/jukebox/sut/JukeBox.Java. This is the implementation of the credits-calculation of the SUT.
- Create in the Tutorial/src folder a folder/package named jukebox.fixtures.
- Create in the package/folder a Java class named CreditsForPayment.Java which contains the following:

```
package jukebox.fixtures;
import jukebox.sut.*;
public class CreditsForPayment {
  private double payment;
  private int credits;
  public void setPayment(double payment) { // setter method
      this.payment = payment;
  }
  public void execute() { // executed after each table row
      this.credits = new JukeBox().calculateCredits(payment);
  }
  public int credits() { // returning function (question mark)
      return this.credits;
  }
}
```

- And compile it as well:
  - Netbeans: Run - Build Project
  - No IDE: run the following in the Tutorial/src-folder (both compile SUT as the fixtures):
    Javac -classpath .;../FitNesse-standalone.jar jukebox/sut/*.Java
    Javac -classpath .;../FitNesse-standalone.jar jukebox/fixtures/*.Java
- This class above is the corresponding fixture class for the FitNesse test page.
- When running the FitNesse test it will search for the fixture called CreditsForPayment, and then will do this for each row:
  - First the setters are called (in this case setPayment(..) function) so the fixture has the input,
  - Then the execute()-function is called to do call the function of the SUT with the given inputs and stores the result in the fixture
  - Then the result is retrieved from the Fixture (in this case by the credits()-function). The FitNesse Test Page compares this value with the given value in the table.
- Finally, the FitNesse test page needs to be configured before the test will work.

Configuring the FitNesse Test Page

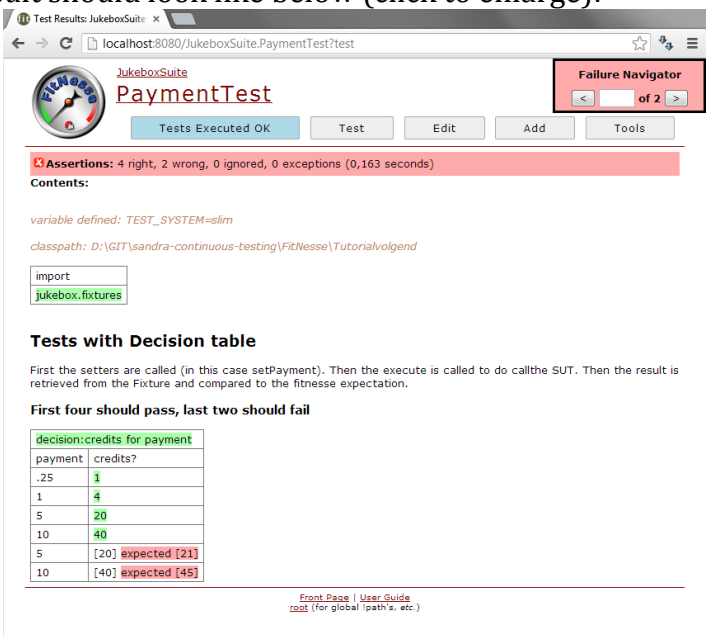
- go to http://localhost:9090/JukeboxSuite.PaymentTest and click on edit.
- Add on the top of the page (just below the "!contents ..."):

```
!define TEST_SYSTEM {slim}

for using Netbeans with standard settings:
!path ../Tutorial/build/classes

|import |
|jukebox.fixtures|
```

- Different IDE's use different output-paths for their .class files. You could change the path to where your .class files are.
- In the case study's project, we will use the NetBeans standard settings. Note: To let it work on all operating systems, the path needs to be specified with "/" (instead of "\")
- This configuration will tell FitNesse:
    - we want to use SLIM-fixtures instead of the default FIT (as mentioned in the introduction)
    - the class path to your project (this is a relative path from the FitNesse-standalone.jar to the class-files)
    - the location of the fixtures for this test (relative from the class path) and that these should be imported (this table is called an import table)

Run the test!

- Go to http://localhost:9090/JukeboxSuite.PaymentTest
- Click on Test in the menu at the right.
- The result should look like below (click to enlarge):



- Note that you could also run the whole test suite, by going to http://localhost:9090/JukeboxSuite?suite and click on Suite. This
- will run all test pages in the suite and give the results.
- Note that normally, the two failed test won't be here as such, cause you could test the correct working with the two above. These are added so you could see how failed tests look like.

4. Test with a Query Table

- *You may skip this step and go to step 5, since we will probably not use it*
- Query tables are, as the name implies, meant to query for data. There are currently 3 kinds of query tables, which are almost identical, but with
- some notable exceptions.

Fixture	Description
Query	A standard query table, which compares the complete set of data in and unordered way.
Subset query	Only those rows defined in the table need to be in the Fixture result.
Ordered query	The order of the rows in the table must be in the same order as the rows returned by the query

- A query table is used to compare the results of a query. This is helpful when you only need to make assertions about data, instead of also manipulating data in the system. The following example only illustrate the first Query-fixture, the others are similar.

Create the test

- Consider line 5-9 and 15-21 from jukebox.sut.JukeBox.Java. The jukebox has a list of songs, songs can be added and you could get a list of songs for a given artist.
- We will create a test that tests the feature of getting a list of songs for a given artist.
- Add the following on the http://localhost:9090/JukeboxSuite.PaymentTest-page by editing it and add it below the Disicion-table tests:

```
!2 Tests with a Query table
first artist misses Zeppelin, second duration is 2:25,
third is not in jukebox, there is an extra
|Query:songs from artist          |Led Zeppelin |
|title              |artist       |duration     |
|Stairway to Heaven |Led          |8:36         |
|Immigrant Song     |Led Zeppelin |2:00         |
|I Dont Exist       |Hiding Band  |0:00         |
```

- This test will try to find the fixture called SongsFromArtist and finds a function that returns a list of rows. Each row returned by the query method is a list of fields. Each field is a two-element list composed of the field name and its value as a String.
- Each row in the table is checked to see if there is a match in the query response. The results of the comparison are colored accordingly, and are checked for extra or missing records. The order of the rows is irrelevant in this query table.

Create the fixture

- Create in jukebox.fixtures Java class named SongsFromArtist.Java which contains the following::

```
package jukebox.fixtures;
import static util.ListUtility.list;
```

```
import Java.util.*;
import jukebox.sut.*;
public class SongsFromArtist {
  String artist;
  public SongsFromArtist(String artist) {
      this.artist = artist;
  }
  public List<Object> query() {
      List result = new ArrayList();
      for (Song song : JukeBox.findSongsFromArtist(artist)) {
       result.add(
        list(
        list("title", song.getTitle()),
        list("artist", song.getArtist()),
        list("duration",song.getDurationInUserFriendlyFormat())
        )
       );
      }
      return result;
  }
}
```

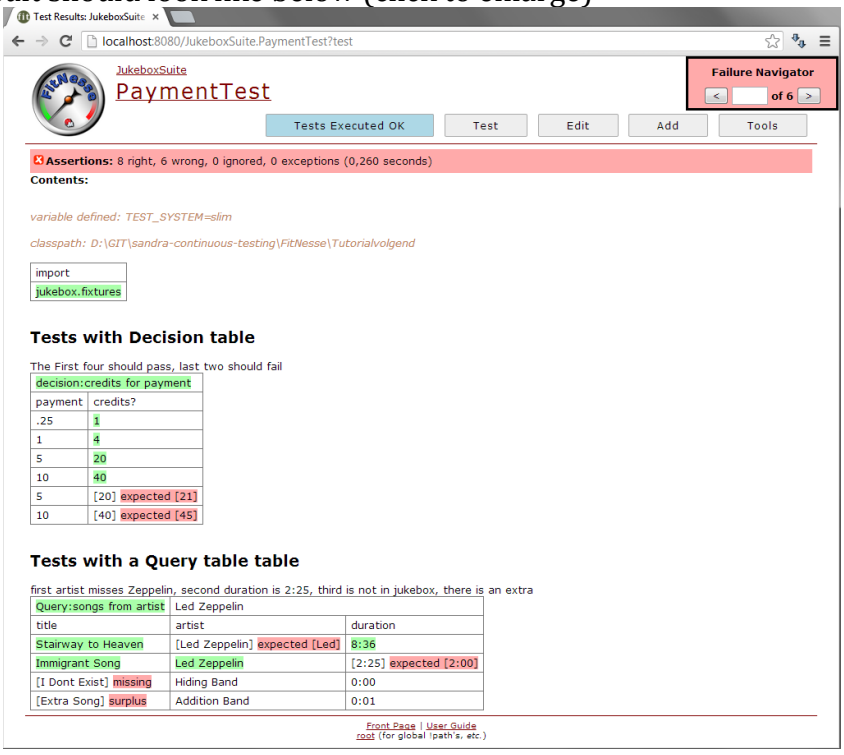
- Note that the list function simply builds an ArrayList from its arguments. It's in the ListUtility class, which is included in the FitNesse.jar.
- Compile the fixture or build the project in the IDE

Configuring the FitNesse Test Page

- The test page is already configured with our first test (see the Configure-step in: 3. Test with a Desicion Table

Run the test!

- Go to http://localhost:9090/JukeboxSuite.PaymentTest
- Click on Test in the menu at the right.
- The result should look like below (click to enlarge)



5. Test with a Script Table

- Script tables are one of the most flexible table styles and can be used for scenario or story based testing. When using a Script table, each statement in the FitNesse test will refer to a method of the fixture used or to an earlier defined scenario. Each statement can be prefixed by one of the Script Table keywords (see below).

Create the test

- We don't have a SUT for this test, we just show how the test page calls the fixture, the fixture then could call the SUT as in the previous two examples.
- We will create a test that tests the feature of depositing and withdrawing money from an account. Add the following on the http://localhost:9090/JukeboxSuite.PaymentTest-page by editing it and add it below the Query-table tests:

```
!2 Tests with script table
|script    |current account                          |
|check     |cash balance should be |0.0              |
|deposit   |0.25                                     |
|check     |cash balance should be |0.25             |
|deposit   |0.75                                     |
|check     |cash balance should be |1.0              |
|$balance= |total deposits                           |
|ensure    |withdraw               |1.0              |
|note      |account should not allow negative balance|
```

- This test will try to find the fixture called CurrentAccount.
    - The first row finds a function called cashBalanceShouldBe() and compares the value with the given value.
    - The second row finds a function called deposit(xxx) and calls it with the given value
    - The $balance row finds a function called totalDeposits() and sets this value in the parameter $balance.
    - The ensure row finds a function called withdraw(xxx) and calls it with the given value. the withdraw(xxx) function gives a boolean back and the ensure expects a true-value in order to pass.
    - The note row does nothing, it is there as an comment

Create the fixture

- Create in jukebox.fixtures Java class named CurrentAccount.Java which contains the following:

```
package jukebox.fixtures;
public class CurrentAccount {
  public double cashBalance;
  public double totalDeposits;
  public CurrentAccount(){
      cashBalance = 0.0;
      totalDeposits = 0.0;
  }
  public double cashBalanceShouldBe() {
      return cashBalance;
  }
  public double totalDeposits() {
      return totalDeposits;
  }
```

```
    public void deposit(double amount) {
        cashBalance= cashBalance+amount;
        totalDeposits = totalDeposits+amount;
    }
    //only withdraw when enough balance
    public boolean withdraw(double amount) {
        if(amount <= cashBalance){
          cashBalance = cashBalance-amount;
          return true;
        }else{
          return false;
        }
    }
}
```

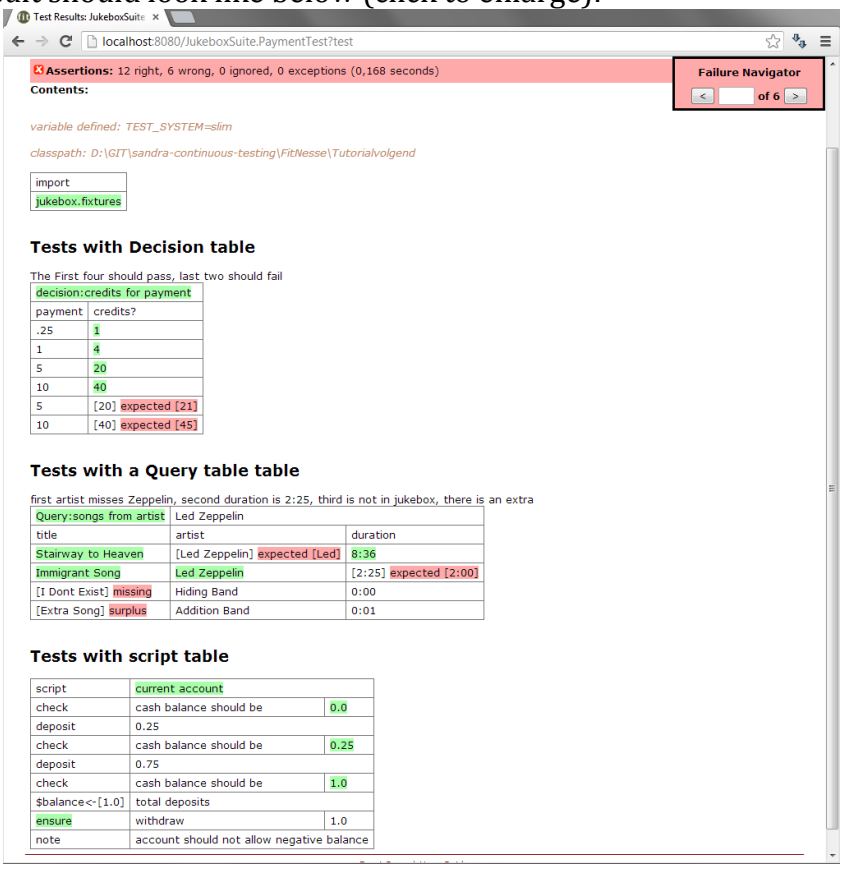
- Here the fixture does all the work, instead of calling an SUT. But you could imagine that the deposit and withdraw functions just call a SUT.
- Compile the fixture or build the project in the IDE

Configuring the FitNesse Test Page

- The test page is already configured with our first test (see the Configure-step in: 3. Test with a Desicion Table)

Run the test!

- Go to http://localhost:9090/JukeboxSuite.PaymentTest
- Click on Test in the menu at the right.
- The result should look like below (click to enlarge):



More info

- More info on script tables: http://FitNesse.org/FitNesse.UserGuide.SliM.ScriptTable

Combining script tables with Scenario tables

- You can also create scenario tables (see http://FitNesse.org/FitNesse.UserGuide.SliM.ScenarioTable)
- Example: you have a scenario for checking if a name is converted well, which shows the original and the normalized name and checks if the result of the normalized name matches the expectations:

```
!|scenario | scenario normalize name | originalName | with
result | normalizedName |
|show | give original back | @originalName|
|show | normalize name | @originalName |
|ensure | normalize original name | @originalName | matches |
@normalizedName |
```

- In a script table, the scenarios can be called. This makes the script more consise:

```
!|script| FixtureClassName
|scenario normalize name| Olàf | with result | Olaf |
|scenario normalize name| Smid-Härt | with result | Smid-Hart |
|scenario normalize name| àáâãäåçý | with result | aaaaaacy |
```

- The result is a concise script-table which can be clicked open for more information (automatically clicks open when there is a fault).

Test:

script	FixtureClassName		
▼scenario normalize name	Olàf	with result	Olaf

scenario	scenario normalize name	originalName	with result	normalizedName	
show	give original back	Olàf			Olàf
show	normalize name	Olàf			Olaf
ensure	normalize original name	Olàf	matches	Olaf	

▶scenario normalize name	Smid-Härt	with result	Smid-Hart
▶scenario normalize name	àáâãäåçý	with result	aaaaaacy

6. Advanced Important FitNesse Info

How to use symbols in SLIM

- FitNesse also supports the use of symbols in their tables (see http://FitNesse.org/FitNesse.UserGuide.SliM.SymbolsInTables).
- Apart from the decision table-example given in the link, symbols can also be used in other tables.
- An example of a usage of an symbol in a script table is shown below:

```
!| script | FixtureClassName |
| $symbolname= | functionReslultString |
| show | $symbolname |
| check | $symbolname | "test" |
| ensure | functionResultBoolInputString | $symbolname |
```

- This is a script table, which, in the first line calls a function that returns a string and puts it's result in a symbol called symbol name
- The second line, displays the value of symbol name when the test is run

- The third line, checks whether the value of the symbol name matches the value "test"
- The last line passes the value of the symbol name to the function specified and calls that function, the result of the function is checked (ensure means that the result needs to be true in order to pass the test).

Other tables in FitNesse, test organization and formatting

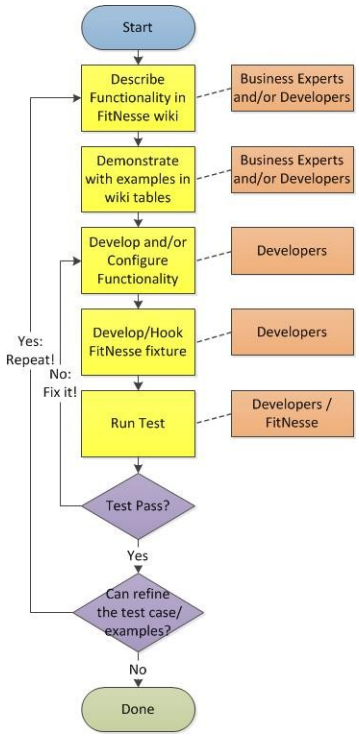
- See the original tutorial (see http://refcardz.dzone.com/refcardz/getting-started-FitNesse) for more information on:
  - Testing with a Library Table
  - How to create a Comment Table
  - How to organize tests
  - Formatting

Using SLIM + More information on several tables and usages

- Although it is mentioned in the tutorial, I would like to mention it again for clarity:
  - We will use FitNesse with SliM-fixtures as an alternative to the FIT-fixtures. See the SliM User Guide (http://www.FitNesse.org/FitNesse.UserGuide.SliM) for more information.
  - The SliM User Guide contains all the possible script tables which can be used under SLiM

Use FitNesse with Test-Driven Development

- Since we will also use test-driven development, we will write tests (and the specifications) before we will code or configure.
- The TDD routine in combination with FitNesse is visualized on the right: describe functionality (business experts BE), demonstrate with examples (BE) - develop functionality (developer) - hook fixture (DEV) - test pass (DEV) – repeat
- Once the first test passes, we write another test, write more code, make the new test run, clean up again and retest. After we have repeated this cycle for all the tests for a specific feature, our work on the feature is done and we can move on to the next feature.



Group tests in test suites

- We will use one main-suite called AllTestSuite. In this suite, multiple other suites are added, based on functionality. So there is a separate suite for testing Aggregations, one for testing Certifications, one for Workflows etc. In these suites, multiple tests are added for different applications and/or different tests.

Use Setup pages

- SetUp is a special page, and is included automatically at the beginning of all test pages in the test suite. Define the paths to the suit-specific fixtures here. Do not mark this page as a test.
- To execute the parent SetUp page in the setup, include it in the sub wiki SetUp with an !include directive, followed by the page name.
- In addition, there is a special root page that defines global definitions for the entire system. Put common HTML content like documentation links into PageHeader and PageFooter, as they are pasted directly into the page code.
- Furthermore, there is an SuiteSetUp page and a SuiteTearDown page which are run before and after the test suite is run. We will use these suites to setup the environment (initialize the sailpoint environment) and tear it down (close connection of SailPoint environment) at the end.

FitNesse in Version Control System, Environments and Automated Builds

- We will use FitNesse in a team by storing tests in our version control system: GIT.
- Tests are plain text files, so modern version control systems can merge most concurrent changes correctly.
- Exclude ErrorLogs and RecentChanges directories (in the wiki) from version control!
- The version control of FitNesse stands in the way, with hundreds of ZIP files.
    - Turn off the FitNesse internal archiving by adding -e 0 to the command used for starting FitNesse (as mentioned in the tutorial as well)
- Developers start FitNesse from the local copy of the repository, and can run test their currently working on locally on their machines and add it to GIT when they are done
- The test-environment will act as the continuous integration server which verifies the builds and runs the tests.
    - A Jenkins server is set up for the continuous build and continuous running of tests
    - Since the integration tests run slowly and ask a lot of resources, we execute them every day on a time that other processes sleep (for instance: at 5 in the morning) instead of after every commit.
    - If the tests fail, developers should be notified. Pre-defined tests are supposed to fail, so these should not notify developers
        - maybe put them in a separate suite?
    - When integrating with continuous build tools, make sure to delete old test results so that they do not get mixed with new ones.
- On this sever, FitNesse is run continuously so that non-technical staff can add their tests to the wiki as well, without all technical work
- One developer is appointed as owner for this process and checks in the above mentioned additions daily in GIT, so that it is merged with the mainline. (however this is automated later on, see next tutorial).

- It is sometimes a good idea to restart FitNesse after tests are updated via Git

How Do we specify tests and use test data?

- We assume dynamic test data: data can and will change, so no assumptions should be made on this
- If possible: undo all changes at the end of your test, so the begin-state and end-state of your test will be the same
- If you add data (for instance: a new user), make sure this happens with a randomized name, so it will not clash with possible other data.

If you like to use custom objects in fixtures

- FitNesse normally uses some standard objects in their tables, but you can use any business domain object as long as it can be uniquely represented by a string. See p111 of the book called "Test Driven .Net Development with FitNesse", which can be downloaded at: http://gojko.net/FitNesse/

7. Using FitNesse with SailPoint

- FitNesse fixtures perform calls to the system under test, where it would need to perform some actions in SailPoint. Currently many test-activities are performed via the web-based GUI. For test automation, we would like to avoid GUI-testing and call to the system directly. Next to the GUI, SailPoint offers an REST API, a console and I created an extract of a deeper functional layer of IIQ, which we will use.

Possible SailPoint Approaches

- GUI
    - GUI tests can be performed. If GUI-testing is ever considered, make sure you use GUI-tests where you can specify commands textual, so you could maintain the code better. A recording option is also preferred, since that lowers the time it costs and the difficulty to create a test. The problem with GUI tests is that GUI's changes more often than the functional level, these changes will break the tests and therefore the tests will lose its reusability and regression testing functionality when it happens, making it less suitable for testing then testing on a functional level
- REST API
    - The REST API (see https://community.sailpoint.com/docs/DOC-1642) is the publicly available API where you can mainly: list identities, create or update them, check an identity's rights and launch workflows (for 6.1: https://community.sailpoint.com/docs/DOC-1668 page 147-202). The API gives results in a JSON-format. The REST API is very limited in the sort of calls it can do to SailPoint, making it not suitable enough for testing
- Console
    - The console makes a live connection to the database of SailPoint IdentityIQ. It is mainly used for debugging and troubleshooting:

testing rules,properties, aggregations, workflows, tasks and performing queries to the database. The console directly talks to the IdentityIQ database, bypassing the GUI. (See the console whitepaper on compass: https://community.sailpoint.com/docs/DOC-1631). Although the console can do a lot more then the api, the console needs to be started from the command line and starting it is very slow. Furthermore the console also does not support everything we would like to do with SailPoint, so this approach is also not suitable enough for testing.

- Our Own SailPointEnvironment
    - Instead of using the console from command line and the public api, there is another way to test on a functional level; we will create our own connection to the SailPoint database. At the start of the test, the SailPoint environment is initialized, which is closed and at the end of the test. During the test, we have access to the SailPointContext (and therefore the private api) and the SailPointConsole (where we can perform calls directly). This approach offers more functionality then the REST-API and console alone, it is faster than the console and on a good functional level. Of course it also has a downside: the lack of documentation of the private API. (see below for more information)

The Chosen SailPoint Approach

- Our Own SailPointEnvironment
    - Using our own SailPointEnvironment is by far the best option, so we will use this option for our tests.
    - As you read above, we will use our own connection to the SailPoint database. I have created a Java class (named SailPointEnvironment) which creates a SpringStarter and a console when it is initialized furthermore you can get and release the SailPoint context from that class.
    - Another class is created, called SailPointConsoleCommands, and has methods for performing commands on the console via the private API and an OutputStream for the feedback. The first version of this class contains a few console commands, which must be extended with more methods that call the console-methods more directly when you need a console-method that isn't already available.
    - With those two base classes, you can create test classes that use the SailPointContext and console commando's. These tests needs to be build with inheritance; for common parts, define a common (abstract) class with functions (either abstract or predefined with a standard body). The example discussed below (see next heading) gives an example on how this inheritance should be done.
    - The created classes for this approach are the following:
        - testing.UsefulFunctionsLibrary
            - It contains usefull generic static methods (not specific to sailpoint).

        - generic static help-methods like: getNumberOfLinesInFile and giveRandomNumberFromRange
    - testing.sailpoint.ConsoleCommands
        - It contains defined static methods for needed console commands
        - static console-command-methods like: runAggregate, runDateAndReturnDate and runConnectorDebug (should be extended with more console-commands when needed)
    - testing.sailpoint.SailpointEnvironment
        - It contains methods for the creation and termination of the SailPoint environment.
        - methods like: constructor, initialize, closeEnvironment, getConsole, giveContext and releaseContext.
        - Note that when you get the context (via giveContext), you should always release the context afterwards, otherwise it will cause errors!
        - Note: in SuiteSetUp of the AllTestSuite, the SailpointEnvironment.initialize-method needs to be called to set the environment before all tests are run
        - Note: in SuiteTearDown of the AllTestSuite, the SailpointEnvironment.closeEnvironment needs to be called to close the environment at the end of all tests
    - testing.sailpoint.fixtures.Test
        - It contains the basis of each test-fixture, right now it is empty.
        - Right now it is empty, it might have some use for the future.
        - Note that all test-fixtures should extend this class (or a child-class of this class)!

Example and more detail and downloads

- I have created an example using this files for two Aggregations.
- Please look into this example, it explains it the above in more detail and also offers a download for these Java-files.
  <<Example Omitted from this attachment: too project-specific>>

Appendix F: Automatic pull/add/commit/push Tutorial

This appendix shows the automatic pull/add/commit/push tutorial that can be followed in order to use the strategy. Text in black are steps to be taken and steps in gray give extra information.

Tutorial – Configure automatic pull, add, commit and push for running FitNesse wiki

- Since we run the FitNesse wiki separately on the test server, such that business users can create test cases without checking out git and running FitNesse for themselves, we need to make sure that the changes that business users make to the wiki are added to Git.
- Furthermore, GIT needs to pull occasionally in order to stay up-to-date.

Options

- There are several options:
  - use GitHub service hooks (source 1: http://net.tutsplus.com/tutorials/other/the-perfect-workflow-with-git-github-and-ssh/ , source 2: http://www.bram.us/2012/05/06/automatic-website-publishing-with-git-github-style/)
  - use Directory Monitor (source 1: http://www.deventerprise.net/DirectoryMonitor, source 2: http://stackoverflow.com/questions/420143/making-git-auto-commit)
  - use Windows-task with script (source: http://www.thehelper.net/attachments/git-tutorial-auto-commit-pdf.18221/) (or use unix cronjob with the same script if you have a unix environment)
  - use Jenkins (source: https://groups.google.com/d/msg/git-users/9wVCXrZabCE/r3ip0XyM_ZIJ)

Chosen Option: use Windows-task with script

- -- Install Git Bash
  - We use Git Bash, so download Git Bash from https://code.google.com/p/msysgit/downloads/list?q=full+installer+official+git

- -- Fix environment
  - add git environment variable (e.g., C:\Program Files (x86)\Git\bin) to the path environment variable of windows.
    - With this, we can run git-commands from cmd, which the windows-task does.
  - If ssh is not enabled (which was the case at my case study):
    - clone the git repository with the command (change it to your own https-directory and add username and password

at user /password for the FitNesse/Jenkins specific user of your repository(which has write-access)):
        - Cone git with password (take the https-link from bitbucket and add username pass)
            - git clone https://user:password@github.com
    - if ssh is enabled (which is more neat):
        - If ssh is enabled (instead of https), you can create a ssh-key instead of giving the password at the git clone (source: https://confluence.atlassian.com/display/BITBUCKET/Set+up+SSH+for+Git):
        - Fix ssh-key:
            - start Git Bash
            - type ssh-keygen -t rsa -C me@email.com (Note: use the same email as the FitNesse/Jenkins account is used)
            - enter enter enter (Note: no passphrase)
            - add the content of: .ssh/id_rsa.pub to the bitbucket-account for the FitNesse/Jenkins-account under his ssh keys.
            - move .ssh folder and its content to:
                - C:\Program Files (x86)\Git\.ssh
            - Clone git with ssh key (take the ssh-link from bitbucket)
                - git clone git@github.com:username/repositoryname.git

- -- Create Autocommit-script
    - create a script called: autocommit.bat which contains:
    - Make sure you change the path of the first cd-statement to the path of your repository you just cloned, including to the FitNesseRoot-directory, so only changes in that directory are added/pushed.

```
:: go to right folder
cd ......../ IdentityIQ/FitNesseRoot
:: first pull
git pull
:: add everything from this folder
git add -A .
:: commit it
git commit -m "automatic commit from FitNesse"
:: push to master
git push origin master
```

- -- Create Windows Task
    - open control panel - task scheduler
    - click on create task
    - general tab:
        - give it a descriptive name and description (for instance: automatic push FitNesse wiki)
    - trigger tab:

        - click new, click on startup, run every hour indefinitally, click on stop task if it runs longer than X and choose 30 minutes.
    - actions tab:
        - browse to the bat-file you just created.

- -- Done!
    - Then you're done! If you start FitNesse from that cloned repository, all changes will be pulled, added, committed and pushed.

Appendix G: Jenkins Tutorial

This appendix shows the Jenkins tutorial that can be followed in order to use the strategy. Text in black are steps to be taken and steps in gray give extra information.

Tutorial - Jenkins: Installing, Use etc

- Jenkins is used as our Continuous Integration Server. We use Jenkins to pull changes from the GIT repository, building the application and FitNesse-tests and then running the unit tests and the FitNesse tests every day (the schedule is configurable).
- The configurations below includes some specific IdentityIQ-parts (for building and deploying IdentityIQ), these parts can be replaced with building and deploying other applications; the way of installing Jenkins and specifying how Jenkins should execute and report FitNesse tests stays the same.

Install Jenkins

- Download the Jenkins war-file from: http://Jenkins-ci.org/ (see the menu on the right)
- We use Tomcat to run our apps, so we deploy in this manner:
  - Move the de war-file to $TOMCAT_HOME/webapps
  - Go to tomcat, and then click on Tomcat Manager
  - In this window, Deploy de jenkins.war
- Go to: http://localhost:8080/Jenkins/, this will start Jenkins.

Install Jenkins Plugins

- Go to Jenkins - Manage Jenkins - Manage plugins - Available
- Install these plugins: Git Plugin; Hudson FitNesse plugin;

Configure Jenkins

- Go to Jenkins and click on Manage Jenkins - Configure System
- check if the path to git.exe exists, if not: fill in the path to git.exe
- at Ant, click Add Ant, give the name apache-ant-<version>

Bug-fixing configurations

- When configuring FitNesse-tests, we experienced this bug (https://issues.Jenkins-ci.org/browse/JENKINS-16204), where the default JDK selection does not work and let the run fail. We used the suggested solution (configuring 2 JDK's and choose one) which worked.
- So install an extra JDK on the machine.
- Go to Manage Jenkins - Configure System
  - at JDK, click add JDK and give it the name JDK<version> and specify the JAVA_HOME of these version (i.e., E:\ProgramFiles\Java\JDK1.7.0_25)
  - Do the same for an older version of Java

Create the Jenkins Job that pulls from git, builds the software, execute tests and reports the results.

- Click on New Job
- name the job "build-war" and choose build a free-style software project
- Project
    - Check Discard Old Builds and choose the #days to keep builds and max builds to keep (for instance 5)
    - Choose the newest JDK version
- Source Code Management
    - Check Git and set up Git via the repository url
- Build Triggers
    - Check Build periodically and fill in the Schedule-textbox the time of the day to start the job, for instance, to start it at 7 in the morning: 0 7 * * * (or use the GIT as trigger)
- Build Steps
    - Click on Add build step - invoke ant and click on advanced
        - Choose the ant version
        - set its target (i.e., clean war prepare.extract.for.console) (for FitNesse tests, the prepare.extract.for.console sets the iiq.properties so that FitNesse can access the database, like it is needed for the console)
        - set the link to the build file (i.e., IdentityIQ/build.xml)
        - set the properties for the build file (i.e., environment=test)
    - Click on Add build step - execute FitNesse tests
        - Choose Start New FitNesse Instance as part of build
        - Set Java working directory on IdentityIQ and set the path of FitNesse.jar and FitNesseRoot, set the port on a unused port (i.e., 9000), set the target page on the main suite (i.e., AllTestSuite) and check it is a suite. Set the HTTP Timeout to 300000 and set the FitNesse results path to IdentityIQ/build/FitNesse-results.xml
- Post-Build Actions
    - Click on Add post-build action - Archive the artifacts
        - set Files to archive: IdentityIQ/build/deploy/*
    - if you have unit tests: Click on Add post-build action - Publish JUnit
        - test result report test report XMLs: IdentityIQ/build/*.xml
    - Click on Add post-build action - Publish FitNesse results report and set the path to FitNesse results file (same as the one above)
- This should be it to automate the execution of the build and tests steps!

Add security

- Without configuring some security, every user that has access to the link has admin-rights to Jenkins, which needs to be managed
- Click Manage Jenkins - Configure Global Security
    - Check enable security
    - Check Access Control - Jenkins's own user database
    - Save (first we need to add users to the user database before restricting more)
- Click Manage Users

    - Create User and add your own credentials
- Try to log in with your just created account
    - If it works, you can enhance security:
        - Click manage Jenkins - Configure Global Security
            - Uncheck Allow users to sign up
            - Check Logged-in users can do anything
            - Save
    - Add other user accounts for your team members.
        - They can change their password after logging in, clicking on their name on the right and then clicking on configure