



# APPLICATION FOR SYNCHRONIZATION OF EVENTS BETWEEN VERSIONONE AND ALM

## Diploma thesis

*Study programme:* N2301 – Mechanical Engineering  
*Study branch:* 3902T021 – Automated Control Systems  
*Author:* **Bc. Michal Říčan**  
*Supervisor:* Ing. Michal Moučka, Ph.D.



## ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Michal Říčan**  
Osobní číslo: **S12000463**  
Studijní program: **N2301 Strojní inženýrství**  
Studijní obor: **Automatizované systémy řízení ve strojírenství**  
Název tématu: **Aplikace pro synchronizaci událostí mezi VersionOne a ALM**  
Zadávací katedra: **Katedra výrobních systémů a automatizace**

### Z á s a d y   p r o   v y p r a c o v á n í :

1. Navrhněte strukturu aplikace pro synchronizaci událostí mezi aplikacemi VersionOne a ALM. Pro přístup k informacím o událostech navrhněte klienta schopného komunikovat s REST API.
2. Navrhněte strukturu pluginu pro Jenkins pro odesílání výsledků testů do ALM.
3. Vše naprogramujte ve vhodných programovacích jazycích.
4. Ověřte funkčnost a správnost vašeho řešení.

Rozsah grafických prací: dle potřeby  
Rozsah pracovní zprávy: cca 45 stran + přílohy  
Forma zpracování diplomové práce: tištěná/elektronická  
Seznam odborné literatury:

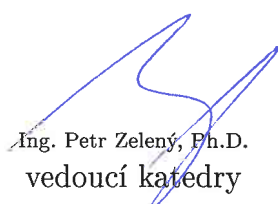
- [1] BECK, K. *Test Driven Development: By Example*. Boston (USA): Addison - Wesley, 2003. ISBN 0-321-14653-0.
- [2] DUVALL, P. M., S. MATYAS a E. GLOVER. *Continuous Integration: Improving Software Quality and Reducing Risk*. Boston (USA): Addison-Wesley, 2007. ISBN 978-0-321-33638-5.
- [3] Hewlett-Packard Development Company. HP ALM Synchronizer User Guide. 2011.
- [4] Hewlett-Packard Development Company. HP ALM OTA a REST reference. 2011.
- [5] NAGEL, CH., B. EVJEN, J. GLYNN a M. SKINNER a K. WATSON. *C# Programujeme profesionálně 2008*. Praha: Computer Press, 2013. ISBN 978-80-251-2401-7.
- [6] *VersionOne User Guide for Enterprise and Ultimate*. (accessed Oct 29, 2014). [online]. [cited 29 Oct 2014]. Available from: [http://community.versionone.com/Help-Center/Getting-Started-Guides/User\\_Guide\\_for\\_Enterprise\\_and\\_Ultimate](http://community.versionone.com/Help-Center/Getting-Started-Guides/User_Guide_for_Enterprise_and_Ultimate).
- [7] *VersionOne REST* (accessed Oct 29, 2014). [online]. [cited 29 Oct 2014]. Available from: <http://community.versionone.com/Developers/Developer-Library/Documentation/API>.

Vedoucí diplomové práce: Ing. Michal Moučka, Ph.D.  
Katedra výrobních systémů a automatizace

Datum zadání diplomové práce: 5. března 2015  
Termín odevzdání diplomové práce: 5. června 2016

  
prof. Dr. Ing. Petr Lenfeld  
děkan



  
Ing. Petr Zelený, Ph.D.  
vedoucí katedry

V Liberci dne 5. března 2015

## Declaration

I hereby certify that I have been informed the Act 121/2000, the Copyright Act of the Czech Republic, namely § 60 - Schoolwork, applies to my master thesis in full scope.

I acknowledge that the Technical University of Liberec (TUL) does not infringe my copyrights by using my master thesis for TUL's internal purposes.

I am aware of my obligation to inform TUL on having used or licensed to use my master thesis; in such a case TUL may require compensation of costs spent on creating the work at up to their actual amount.

I have written my master thesis myself using literature listed therein and consulting it with my thesis supervisor and my tutor.

Concurrently I confirm that the printed version of my master thesis is coincident with an electronic version, inserted into the IS STAG.

Date:

Signature:

## Acknowledgment

It would not have been possible to write this diploma thesis without the help and support of the kind people around me, to only some of whom it is possible to give particular mention here.

First of all I would like to thank to my consultant Srdjan Nalis (Mr. Automation) for his timely advice, meticulous scrutiny, support and friendship. I'm pleased to cooperate with someone skilled like he is.

Also I would like to thank to my mentor Ing. Michal Moučka, Ph.D., for his scholarly advice and advices regarding the processing of the thesis.

Last but not least I would like to thank to my whole family for their never ending support not just in the school times. There is no room to give particular mention to every one as my family is pretty big. But I would like to mention one person which is my girlfriend Markéta Pipková which is really tolerant to my coding passion and also gave me priceless support in the times when I was working on the thesis.

## ANOTACE

Diplomová práce se zabývá tvorbou softwarových aplikací zlepšujících SDLC proces (Software development life cycle). Teoretická část je věnována vybraným metodikám vývoje softwaru, přináší pohled na evoluci těchto metodik a jejich srovnání. Dále jsou v teoretické části shrnuty klíčové vlastnosti nástrojů, pro které byly aplikace vyvíjeny. Větší část práce je věnována praktické části, kde pro každou vyvíjenou aplikaci jsou stručně popsány nejdůležitější moduly a komponenty, včetně popisu chování těchto komponent.

**Klíčová slova:** metodika vývoje softwaru, agilní, Version One, Jenkins, HP Application Lifecycle Management, REST API, synchronizátor, plugin, události

## ANOTATION

Diploma thesis deals with creation of software applications which improve SDLC process (Software development life cycle). Theoretical part is devoted to selected methodologies of software development also brings view to evolution of those methodologies and their comparison. The thesis then summarizes key features of tools for which those applications were developed. Most of the thesis is devoted to practical part where for each of applications is brief description of most important modules and components including a description of functionality.

**Keywords:** software development methodologies, agile, Version One, Jenkins, HP Application Lifecycle Management, REST API, synchronizer, plugin, events

# Table of contents

List of abbreviations .....	9
Introduction .....	10
1 Development methodologies.....	11
1.1 Waterfall model.....	11
1.2 Agile model .....	12
1.3 Agile vs. Waterfall Development Process .....	15
1.4 Continuous Delivery .....	18
1.5 Current Problems and Constrains .....	20
2 HP ALM (Application Lifecycle Management).....	23
3 VersionOne (V1) .....	26
4 Jenkins .....	28
5 V1/ALM Synchronizer.....	31
5.1 Research .....	31
5.1.1 Limitations and bottlenecks.....	32
5.1.2 How to capture event on REST? .....	32
5.2 Architecture .....	33
5.3 REST Client .....	35
5.3.1 Version One REST Client .....	35
5.3.2 Application lifecycle management REST Client .....	40
5.4 Factories .....	47
5.4.1 Requirement factory.....	47
5.4.2 Defect factory .....	49
5.5 Synchronizer configuration .....	50

5.5.1	General information .....	51
5.5.2	OAuth2 settings .....	52
5.5.3	Project linkage.....	54
5.5.4	Entities customization.....	55
5.5.5	IDs and Requirements mapping.....	57
5.5.6	Subscribers.....	57
5.5.7	Summarization of the configuration.....	58
5.5.8	Read/Write of configuration file.....	59
5.5.9	Password encryption/decryption manager .....	60
5.6	Synchronizer core .....	60
5.6.1	Initializer Service .....	61
5.6.2	V1 Listener .....	63
5.6.3	ALM Listener .....	66
5.6.4	Controller .....	67
5.6.5	Mapper Service .....	71
5.6.6	Verify Service .....	73
5.6.7	Mail Service .....	74
5.6.8	Repository .....	75
5.6.9	GenericObject .....	76
5.6.10	Workflow .....	77
5.7	Synchronizer instance manager .....	78
5.7.1	Instance Process.....	78
6	Jenkins plugin – Dingo .....	82
6.1	Research .....	82
6.2	Architecture .....	83
6.3	Pre-defined structure .....	83



6.4	Dingo Core .....	86
6.4.1	ALM Client .....	87
6.4.2	ALM Factories .....	88
6.4.3	ALM Entities .....	90
6.4.4	ALM Parser .....	92
6.4.5	Configuration .....	93
6.4.6	Logger.....	94
6.4.7	Common entities.....	94
6.4.8	Common handler .....	96
6.4.9	JUnit entities .....	97
6.4.10	JUnit handler .....	97
6.4.11	NUnit entities .....	99
6.4.12	NUnit handler .....	99
6.4.13	Push service.....	100
6.5	Jenkins Dingo .....	102
6.5.1	Jelly config.....	102
6.5.2	Dingo plugin controller .....	104
	Conclusion .....	106
	References.....	107

## List of abbreviations

**HP** – Hewlett-Packard

**ALM** – Application Lifecycle Management

**V1** – Version One

**SAFe** – Scaled Agile Framework

**REST** – Representational State Transfer

**OTA** – Open Test Architecture

**API** – Application Programming Interface

**SaaS** – Software as a Service

**AQMS** – Automation & Quality Management Symposium

**CRUD** – Create, Read, Update and Delete operations

**SSO** – Single Sign-On

**HTTP** – Hypertext Transfer Protocol

**HTTPS** – Hypertext Transfer Protocol Secure

**UI** – User Interface

**SAML** – Security Assertion Markup Language

**XML** – Extensible Markup Language

**URL** – Uniform Resource Locator

**JSON** – JavaScript Object Notation

**LINQ** – Language Integrated Query

**KPI** – Key performance indicator

**RC** – Return code

**DAO** – Data access object

**YAML** – Ain't Markup Language

**MQAT** – Mainframe Quality Automation Team

**JUnit, NUnit, xUnit** – Unit testing frameworks

**SCM** – Source Code Management

**SAX** – Simple API for XML

# Introduction

Since the early days of software development, all the IT companies tried to answer the following questions:

1. How do I deliver software that customers will use / need
2. How do I deliver software of the highest quality
3. How do I deliver software before the competition

There have been many methodologies and **SLDC** processes (the **S**ystems **D**evelopment **L**ife **C**ycle, also referred to as the application development life-cycle, is a term used in software engineering to describe a process for planning, creating, testing, and deploying an information system) put forward to address these questions. The methodology that prevailed and was used (until recently) as a standard for all software development companies was called “**Waterfall**”

The brief entertainment of selected development methodologies will be described at the theoretical part, and also selected tools which helps you to follow those methodologies in most efficient way.

As the second part of the thesis will be developed software which helps companies working by agile methodologies to hook up information from tools. The first software should hook up information between project management tool and global test management tool. The second software will shares results from CI server tool to global test management tool, because of linked information we gets better overview about project health and we will be able to use reporting features from the tools to track project health easily.

# 1 Development methodologies

## 1.1 Waterfall model

The Waterfall Model was one of the first Process Model to be introduced. It is also referred to as a **linear-sequential life cycle model**. It is very simple to understand and use. In a waterfall model, each phase must be completed fully before the next phase can begin. At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project. In this model the testing starts only after the development is complete. In **waterfall model phases** (Figure 1) do not overlap.

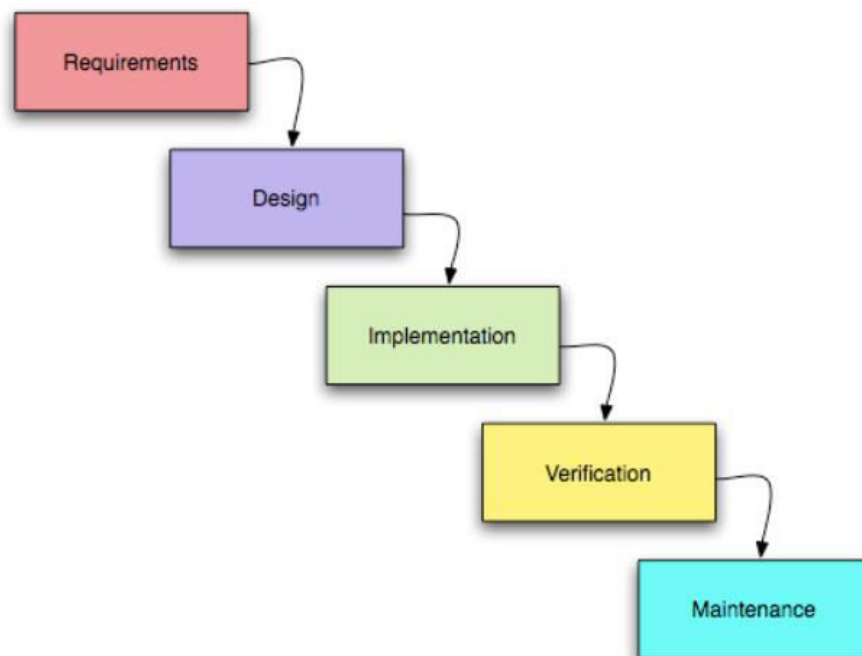


Figure 1 – Waterfall model phase<sup>1</sup>

---

<sup>1</sup> Source: [http://learnaccessvba.com/images/application\\_development/Waterfall\\_model.png](http://learnaccessvba.com/images/application_development/Waterfall_model.png)

Due to ever changing requests and customers' needs there was a problem implementing Waterfall approach to huge scaled projects where requirements and outcomes are not clear from the very start. This opened the doors to new SDLC process to be put forward and the age of “Agile” was born.

## 1.2 Agile model

**Agile development model** (Figure 2) is a type of Incremental model. Software is developed in incremental, rapid cycles (called Sprints). This results in small incremental releases with each release building on previous functionality. Each release is thoroughly tested to ensure software quality is maintained. It is used for time critical applications. **Scrum** is the most widely recognized and adopted agile methodology. Each product team is divided in small operational units called **Scrum teams**.

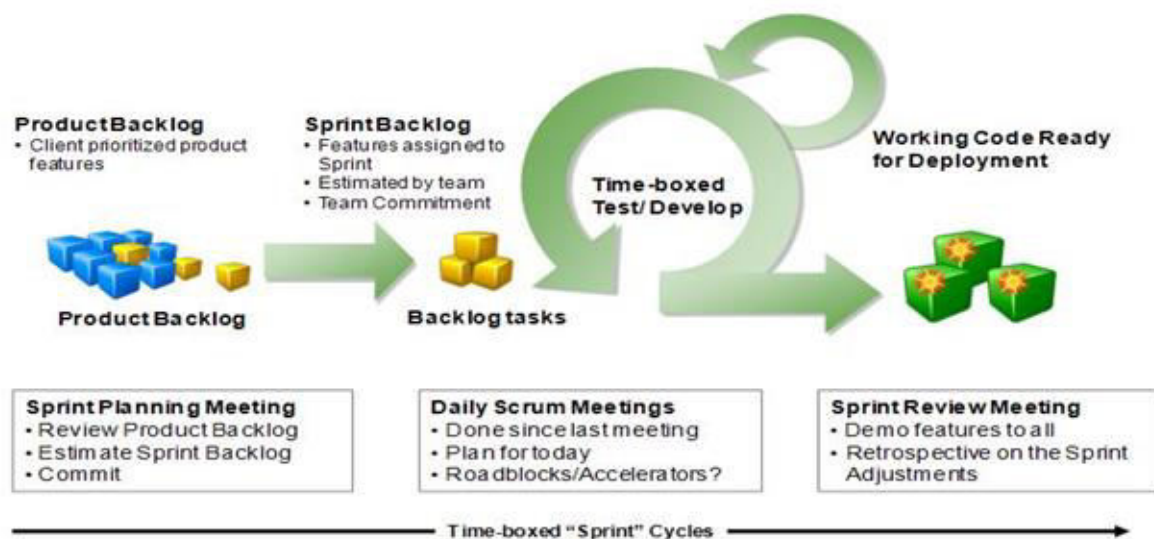


Figure 2 – Agile development model<sup>2</sup>

<sup>2</sup> Source: <http://seyekuyinu.com/file/2011/03/agile-scrum-process.jpg>

## Scrum

The term **Scrum** emerged as a rugby analogy where a self-organizing team moves down the field – together. A key principle of scrum is its recognition that during a project the customers can change their minds about what they want and need, and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, scrum adopts an empirical approach—accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to deliver quickly and respond to emerging requirements.

## Product Backlog

**Product Backlog** is simply a list of items / functionalities that needs to be done within the project or a release. It replaces the traditional requirements specification artifacts.

## Sprint

In Agile work is confined to a regular, repeatable work cycle, known as a **sprints** or **iterations**. **Sprints** used to be 30 days long, but today many teams prefer shorter **sprints**, such as one-week or three-week **sprints**.

## Sprint Backlog

The **sprint backlog** is a list of tasks identified by the **Scrum** team to be completed during the **sprint**. During the **sprint** planning meeting, the team selects some number of **backlog** items, usually in the form of user **Stories**, and identifies the tasks necessary to complete each one.

## Sprint Planning Meeting

During the sprint planning meeting, the **product owner** describes the highest priority features to the team. The team asks enough questions that they can turn a high-level user story of the product backlog into the more detailed tasks of the sprint backlog.

## Daily Scrum Meetings

On each day of a sprint, the team holds a daily scrum meeting called the **daily scrum or daily stand-ups**. Meetings are typically held in the same location and at the same time each day. Ideally, a daily scrum meeting is held in the morning (where all the participants are standing up, hence the name), as it helps set the context for the coming day's work or to resolve the problems occurring during the previous day of the sprint.

## Sprint Review Meeting

When the **sprint** ends, it's time for the team to present its work to the Product Owner. This is known as the **sprint review meeting**. At this time, the Product Owner asks the team to demonstrate a potentially customer shippable product components. The Product Owner declares which items are truly done or not.

## Product Owner

The Scrum product owner is typically a project's key stakeholder. Part of the product owner responsibilities is to have a vision of what is to be building, and convey that vision to the scrum team. The agile product owner does this in part through the product backlog, which is a prioritized features list items for the product.

## Scrum Team

A Scrum team in a Scrum environment does not include any of the traditional software engineering roles such as programmer, designer, tester or architect. Everyone on the project works together to complete the set of work they have collectively committed to complete within a sprint.

## 1.3 Agile vs. Waterfall Development Process

### **Advantages of Agile model:**

- Customer satisfaction by rapid, continuous delivery of useful software.
- People and interactions are emphasized rather than process and tools. Customers, developers and testers constantly interact with each other.
- Working software is delivered frequently (weeks rather than months / year).
- Face-to-face conversation is with customers and stakeholders.
- Close daily cooperation between business people and developers.
- Continuous attention to technical excellence, good design and product quality.
- Regular adaptation to changing circumstances.
- Even late changes in requirements are welcomed

### **Advantages of waterfall model:**

- This model is simple and easy to understand and use.
- It is easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- In this model phases are processed and completed one at a time. Phases do not overlap.
- Waterfall model works well for small projects where requirements are very well understood and are not changing.

### **Disadvantages of waterfall model:**

- Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought, faulty in development or out of concept.
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.



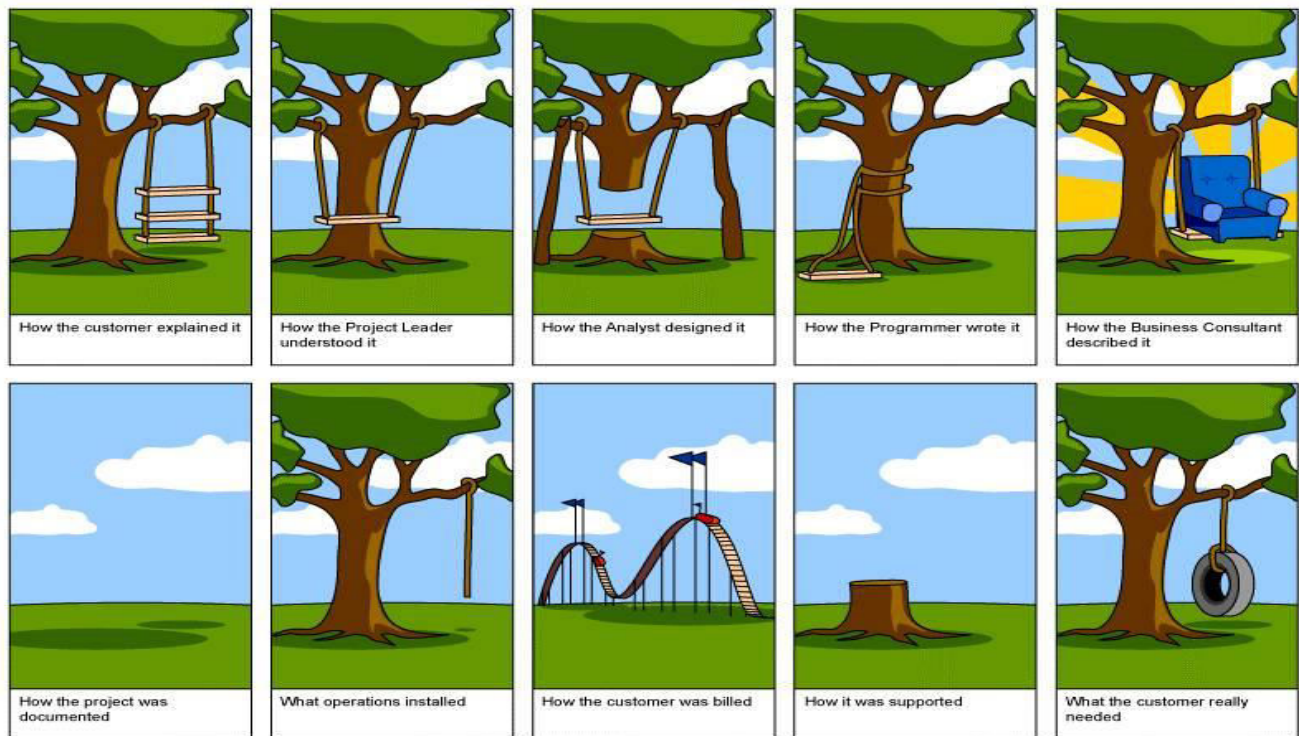
### **When and why o use Agile model:**

- When new changes are needed to be implemented. The freedom agile gives to change is very important. New changes can be implemented at very little cost because of the frequency of new increments that are produced.
- To implement a new feature the developers need to lose only the work of a few days, or even only hours, to roll back and implement it.
- Every deliverable tested to assure the highest quality product reaches the customer. Unlike in Waterfall model where testing is done on the end of development cycle.
- Unlike the waterfall model in agile model very limited planning is required to get started with the project. Agile assumes that the end users' needs are ever changing in a dynamic business and IT world. Changes can be discussed and features can be newly added or removed based on feedback. This effectively gives the customer the finished system they want or need.
- Both system developers and stakeholders alike, find they also get more freedom of time and options than if the software was developed in a more rigid sequential way. Having options gives them the ability to leave important decisions until more or better data or even entire hosting programs are available; meaning the project can continue to move forward without fear of reaching a sudden standstill.

Key takeaway from Agile SDLC approach is:

**“Deliver software quickly with highest quality that customers can use!”**

And we use the Agile SLDC approach to avoid the problems shown at figure 3:



**Figure 3 – Problems of waterfall approach<sup>3</sup>**

As we understood from comparing different SLDC models is that large software development corporations (like CA technologies) must use agile methodology to stay competitive in today's software market. In the following chapters we will focus on how the software can be continuously delivered and how the quality can be achieved during this process.

<sup>3</sup> Source: <https://astheqaworldturns.files.wordpress.com/2011/03/requirements.jpg>

## 1.4 Continuous Delivery

Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time.

You're doing continuous delivery when:

- Your software is deployable throughout its lifecycle.
- Your team prioritizes keeping the software deployable over working on new features.
- Anybody can get fast, automated feedback on the production readiness / quality of their systems any time somebody makes a change to them.
- You can perform push-button deployments of any version of the software to any environment on demand.

You achieve continuous delivery by continuously integrating the software done by the development team, building executables, and running **automated tests** on those executables to detect problems. Furthermore you push the executables into increasingly production-like environments to ensure the software will work in production.

To achieve continuous delivery you need:

- A close, collaborative working relationship between everyone involved in delivery (**DevOps approach**).
- Extensive **automation / automation testing and integrations** of all possible parts of the delivery process, usually using a variety of Continuous Integration / Delivery tools and methodologies.

## What is DevOps approach?

**DevOps** ("development" and "operations") is a software development method that stresses communication, collaboration, integration, automation, and measurement of cooperation between software developers and other information-technology (IT) professionals.

The visualization of DevOps approach is shown at figure 4:

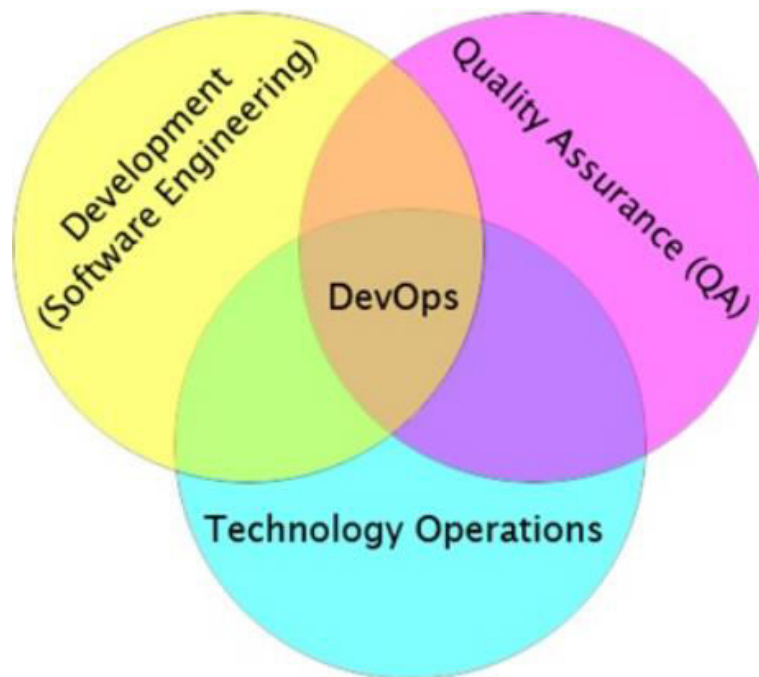


Figure 4 – DevOps approach<sup>4</sup>

---

<sup>4</sup> Source: <http://upload.wikimedia.org/wikipedia/commons/thumb/b/b5/Devops.svg/2000px-Devops.svg.png>

## 1.5 Current Problems and Constrains

As organizations are rapidly changing to Agile methodology (after decades of Waterfall development) many problems, legacy systems / process and other constrains are “creeping” out on daily bases.

One of the biggest problems is how to achieve high quality of a product during rapid development cycles. As we explained earlier testing was a phase that traditionally was done only when development cycles was finished, now in Agile every deliverable component needs to be tests, each integration of the components needs to be tested, each new component needs to be regression (regression testing) tested and etc.

In the Waterfall approach the general consensus is to build comprehensive manual test plans and then have the team of QA engineers execute these (again manually) exercising the software in search of errors and problems. Sometimes in Waterfall approach testing was as long process as development, so testing of software last for months on time.

**Let’s now take an Agile example:** Team is delivering a software functional component in 2 weeks development cycle (Sprint), one of deliverable for that component is a Story that component needs to be fully tested from perspective of functionality and performance. Part from that each stakeholder (Product Owner. Manager, etc.) must be informed about the quality of each impacted Story, so he/she can make informed decision about done criteria for this and other impacted components, eventually about the status of the complete software / application release.

How to achieve that when almost all the testing traditionally is done in long cycles and almost exclusive manually without any centralized repository? Before we answer this question, let’s first understand differences and advantages / disadvantages of manual and automated testing?

## What is manual testing?

Manual Testing is the process of testing software for defects, where testers exercise the software behavior simulating the end user actions, to explain testing coverage; test engineers usually create Test Plans containing a set of important test scenarios (aka. Test Cases) that they will follow during the tests “execution”.

## What is automated testing?

Is use of software to control the test execution. The comparison of actual results to predicted ones, setting up test control and test reporting functions is controlled by automation tool / software. Test automation usually involves automating a manual process already in place (Test Plans). There are three most common types of Automated Tests: Code – driven automation, Headless / API layer automation and GUI (Graphical User Interface) test automation.

## Why Automate?

Manual Testing	Automated Testing
<ul style="list-style-type: none"><li>▪ Time consuming</li><li>▪ Low reliability</li><li>▪ Human resources</li><li>▪ Inconsistent</li></ul>	<ul style="list-style-type: none"><li>▪ Speed</li><li>▪ Repeatability</li><li>▪ Programming capabilities</li><li>▪ Coverage</li><li>▪ Reliability</li></ul>

Figure 5 – Comparison of key attributes of manual/automated testing

There is no question that automation testing needs to be a big part of your Agile process, if nothing else then due to the point that automation testing tools can execute and give accurate result of 1000 of tests in the same time frame than human user can do for 1 test manually.

Now when we decided that automation is the way to go for our Agile project, next step is choosing appropriate tools and frameworks to achieve different levels of product automation

(Functional, Regression, Performance, UI, Web, Client Side, Mobile, Unit, API, etc.), as well as centralized repository where all the data of the testing process will be managed and a way how we are going to incorporate / integrate testing data with backlog deliverable items that are defined in our centralized tool for Agile project planning.

This document will focus on integrations / synchronization between test management and Agile planning tools as used in MFBU (Mainframe Business Unite of CA technologies), and how the gap between their integration is overcome in Continuous Delivery process.

Tool used for Test Management is HP ALM (Application Lifecycle Management), Agile planning tool is VersionOne (V1). The V1/ALM Synchronizer tool is homegrown developed tool (main theme of this master thesis) that made integration/ synchronization between these entities possible.

## 2 HP ALM (Application Lifecycle Management)

HP ALM is web-based global test management solution that helps manage all information about applications releases, testing cycles, requirements, test and defect from a central repository. It manages the entire quality process with built-in traceability

HP ALM streamlines (Figure 6) the testing process—from release and requirements (components) management through planning, scheduling and running tests to defect tracking—in a single browser-based application. HP ALM offers integration with HP automation testing tools as well as third-party and custom testing tools or requirement and configuration management tools. HP ALM communicates seamlessly with the testing tool of choice, providing a complete solution to fully automated application testing.



**Figure 6 – HP ALM streamlines**



## **Release (Testing) Management module**

Release Management module is used to of managing software releases (from quality perspective) from development stage to software release. It is a relatively new but rapidly growing discipline within software engineering.

## **Requirements (Components) module**

Is used to capture, manage and track requirements throughout the development and testing cycle. Its key features are: Capture, manage and track requirements throughout the development and testing cycle, manage different types of requirements, store requirements in a central repository with native version control and base lining capabilities, reuse and share application requirements and manage user stories for agile projects.

## **Test Plan module**

Test Plan module is used to create and store manually or automation tests that will be used to test applications readiness.

## **Test Resources**

Test Resources module enables you to manage resources used by your tests. Organization of resources is by defining a hierarchical test resource tree containing resource folders and resources. In this module we keep our test function libraries, data sheets, parameters, object definitions and more.

## **Test (execution) Lab module**

Is used to create test set that contains a subset of the tests in an ALM project designed to achieve specific testing goals. Run the manual and automated tests from the project to locate defects and assess quality of the release or component.

## **Defect module**

Locating and repairing application defects efficiently is essential to the development process. Using the ALM Defects module, we can report design flaws in the application / component and track data derived from defect records during all stages of the application management process.

## **Dashboard module**

Is used to do the analyze ALM data by creating graphs, project reports, and Excel reports. You can also create dashboard pages that display multiple graphs side-by-side.

HP ALM offers OTA (Open Test Architecture) API architecture that allows customization of components and modules, so that ALM can be tailored to follow individual organization SLDC models or to be integrated / synchronized with any third party Agile planning tools.

Application Lifecycle Management tool is suitable out of the box for any kind of Waterfall or Agile projects, the only predicament lays on customizations and integration that you wish to follow.

### 3 VersionOne (V1)

Version One is an all in one agile project management platform / tool that supports alignment between all three levels of enterprise agile project management (Portfolio, Program, and Team). Built from the ground up to support agile software development methodologies such as Scrum, Kanban, Lean, XP, SAF and hybrid, VersionOne is suite of right-sized product editions help companies scale agile faster, easier, and smarter.

The flow of program and project management is shown at figure 7.



Figure 7 – Program & Project management flow

## **Agile Portfolio Management**

Visualize, manage and report on your strategic, cross-project agile initiatives, keeping business and management priorities aligned with delivery through effective enterprise-wide project management.

## **Product Planning**

Plan and track your agile requirements/components, epics, stories, goals, and defects across multiple projects and teams.

## **Release Planning**

Prioritize, forecast, and report progress on your releases and agile teams in a simple, consolidated drag-and-drop environment. Coordinate multiple teams, increase team member visibility into acceptance and regression testing progress and increase predictability of delivery dates using interactive tools.

## **Sprint Planning**

Iteratively plan user stories, defects, tasks, tests, and impediments in a single environment.

## **Tracking**

Easily track portfolio, project, and Scrum team progress.

## 4 Jenkins

Jenkins is an open source continuous integration tool, released under MIT license, forked from Hudson after a dispute with Oracle. It provides continuous services for software development.

It is server-based system running in a servlet container such as Apache Tomcat and supports SCM tools including CVS, Subversion, Git or RTC. Jenkins is able to execute Apache Ant or Apache Maven base projects as well as arbitrary shell scripts and Windows batch commands.

Builds can be started by various means, including being triggered by commit in a version control system via cron-like system, building when other builds have completed, and by requesting a specific build URL.

Current Jenkins focuses on the following two jobs:

- **Building/testing software projects continuously**, just like CruiseControl or DamageControl. In a nutshell, Jenkins provides an easy-to-use so-called continuous integration system, making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build. The automated, continuous build increases the productivity.
- **Monitoring executions of externally-run jobs**, such as cron jobs and procmail jobs, even those that are run on a remote machine. For example, with cron, all you receive is a regular email that captures the output, and it is up to you to look at them diligently and notice when it broke. Jenkins keeps those outputs and makes it easy for you to notice when something is wrong.

Jenkins offers the following features:

- **Easy installation**, it is distributed as `java -jar jenkins.war` or it is deployed in a servlet container.
- **Easy configuration**, because Jenkins can be configured entirely from web GUI with extensive on-the-fly error checks and inline help.
- **Change set support**, Jenkins can generate a list of changes made into the build from SCM tool, also done in a fairly efficient fashion to reduce load on the repository.
- **Permanent links**, it gives clean and readable URLs for most of its pages, including some permalinks like “latest build”/“latest successful build”, which can be linked from elsewhere.

- **RSS/E-mail/IM Integration**, Jenkins monitor build results and offers to get real-time notification through RSS, E-mail etc.
- **After-the-fact tagging**, build can be tagged long after builds are completed.
- **Junit/TestNG test reporting**, reports can be tabulated, summarized, and displayed with history information, such as when it started breaking etc. History trend is plotted into a graph.
- **Distributed builds**, Jenkins can distribute build/test loads to multiple computer. This lets you get the most of out of those idle workstations sitting beneath developers desks.
- **File fingerprinting**, it can keep track of which build produces which jars, and which builds using which version of jars, and so on. This works even for jars that are produced outside Jenkins, and is ideal for projects to track dependency.
- **Plugin support**, Jenkins can be extended via 3<sup>rd</sup> party plugins. You can write plugins to make Jenkins support tools/processes that you team uses.

One of the biggest advantages of Jenkins compared to others CI servers is the community of developers. To this day Jenkins CI project at GitHub offers 1366 plugin/project and 587 developers working on it.

On the figure 8 Is shown the simple workflow of CI process with usage of Jenkins.

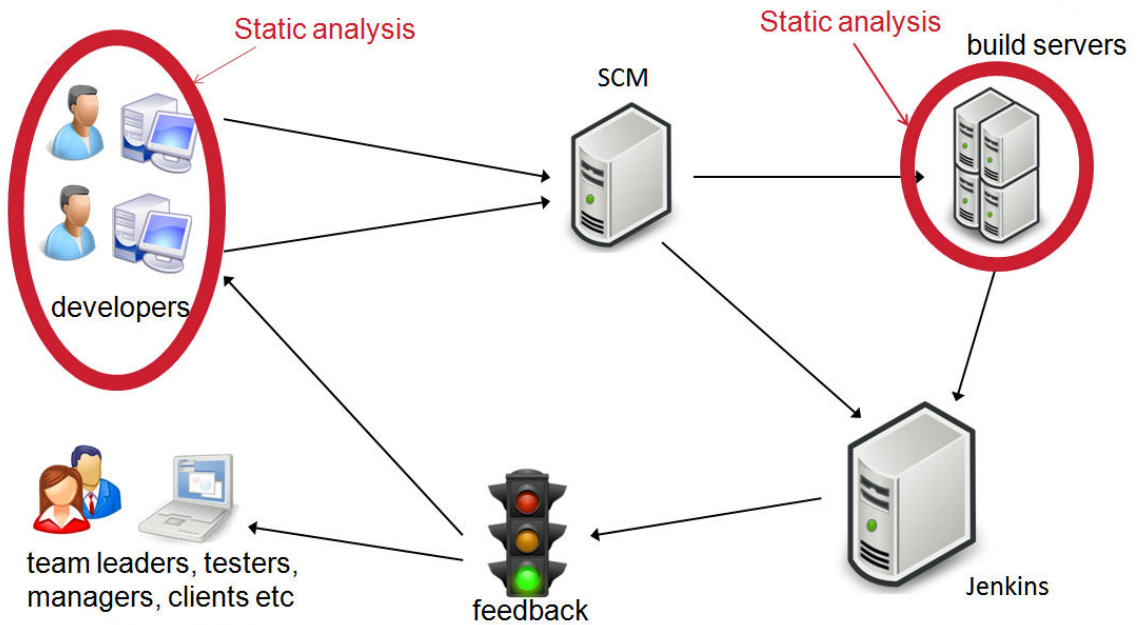


Figure 8 – CI workflow process <sup>5</sup>

---

<sup>5</sup> Source: [http://1.bp.blogspot.com/-fwJu25d\\_4YQ/Up0u3Irlr4I/AAAAAAAAA6s/Z3pIhZb\\_Ag/s640/Git-WorkFlow-Part3.JPG](http://1.bp.blogspot.com/-fwJu25d_4YQ/Up0u3Irlr4I/AAAAAAAAA6s/Z3pIhZb_Ag/s640/Git-WorkFlow-Part3.JPG)

## 5 V1/ALM Synchronizer

On building this software I went through all stages of software development process. First of all research needs to be done, then build prototype based on specifications and knowledge which I get from research part. Then I tried to run that prototype on development environments with sandbox projects. As I was sure that prototype is working for me I send it to chosen teams inside company to get their feedback. After I get feedback I can start with expanding of prototype. Every step at process was also discussed with my consultant Srdjan Nalis.

### 5.1 Research

At research part I started learning both tools more deeply. Firstly from the perspective of user and then from perspective of a developer. From developer point of view I was most interested in data manipulation. VersionOne offers only REST API or a direct connection to database. HP Application Lifecycle Management offers REST API, OTA API and also direct connection to database.

From options mentioned above for both tools I chose the REST API. For the Version One mainly because direct connection to database needs special permission and VersionOne is hosted as SaaS application, which means that application (database) is not hosted on CA servers. For the ALM the reason why I chose REST API was the information that OTA API is at the end-of-life and should not be available in newer versions of ALM and should be replaced with REST API. Information about end-of-life of OTA API I got directly from HP engineers. Thanks to the CA Technologies I was able to attend AQMS QA Automation symposium Prague 2014 at November where those information were shared.

After I know what can be achieved through REST APIs I need to talk with managers inside company and discuss the software requirements also negotiate some compromises between their expectations and what I'm able to achieve in given time and also knowledge level of technologies, processes etc.



As I was getting deeper and deeper I was able to identify that advanced knowledge of some programming language will be needed. From this perspective I chose C# programming language.

With chosen language comes a restriction for used servers where the synchronizer can run. Because application is written in C# with .NET support it can run only at Microsoft servers.

Based on the research I was able to recognize limitations and also bottlenecks, about I will be talking next.

### 5.1.1 Limitations and bottlenecks

First limitation is caused by chosen APIs because ALM API does not offer any existing REST client, so the creation of client was required. On the other hand I was able to design client exactly to my purpose.

HP ALM REST API is just under development, so there is almost no documentation for it and also provides limited functionality besides OTA API.

Version one offers REST API client only for **rest-1.v1** endpoint and does not support SSO login and OAuth2 which is needed for connection to **query.v1** endpoint. So the creation of client or modification of existing client was required.

As REST is used only for CRUD (Create, Read, Update, and Delete) actions I need to find out a way how to capture events through it as there was no other usable API to go with.

Whole process of synchronization needs to be easily modified by customer needs. Which means to allow customer map existing fields, choose which entities will be shared, define where to be system data for synchronization stored etc.

Software needs to be designed in the easily expandable/modifiable way.

### 5.1.2 How to capture event on REST?

That's the question! But thanks to CA Technologies as subscriber to enterprise edition of VersionOne I was able to talk with developers and service architects which are responsible for

Version One REST API. Thanks to information from them about internal Version One events, processes, workflows and suggestions we managed that there should be chance to poll the history and trigger my pseudo-events.

I found some articles that for the purpose of “scanning” is best long polling technique which means that the client requests information from the server exactly as in normal polling, except it issues it is HTTP/S requests (polls) at a much slower frequency. If the server does not have any information available for the client when the poll is received, instead of sending an empty response, the server holds the request open and waits for response information to become available. Once it does, the server immediately sends an HTTP/S response to the client, completing the open HTTP/S Request. In this way the usual response latency (the time between when the information first becomes available and the next client request) otherwise associated with polling clients is eliminated<sup>6</sup>.

The problem with long polling technique is that VersionOne is running on apache-like servers and their thread-per-request model does not work well with long polling.

So I continued with classic requests which should not slowdown servers too much if requests will be created carefully.

## 5.2 Architecture

Architecture was designed with regards to limitations mentioned above. Software is created by three main modules and those modules are **Synchronizer configuration**, **Synchronizer core** and **Synchronizer instance manager**.

**Synchronizer configuration** allows to user create configuration file, where information like login credentials, URLs of applications, projects for synchronization, mapping of fields and mapping of entities can be stored. This component is designed in the way of wizard UI which will lead user step-by-step for creation of configuration file.

---

<sup>6</sup> Source: [http://en.wikipedia.org/wiki/Push\\_technology](http://en.wikipedia.org/wiki/Push_technology)

**Synchronizer core** is main part/feature, it is driving a synchronization based on specifications from configuration file.

**Synchronizer instance manager** is layer above synchronizer core for presentation of the data from synchronizer process.

Because Synchronizer configuration and Synchronizer instance manager consists mostly from forms the architecture of synchronizer core will be shown only and can be seen on figure 9.

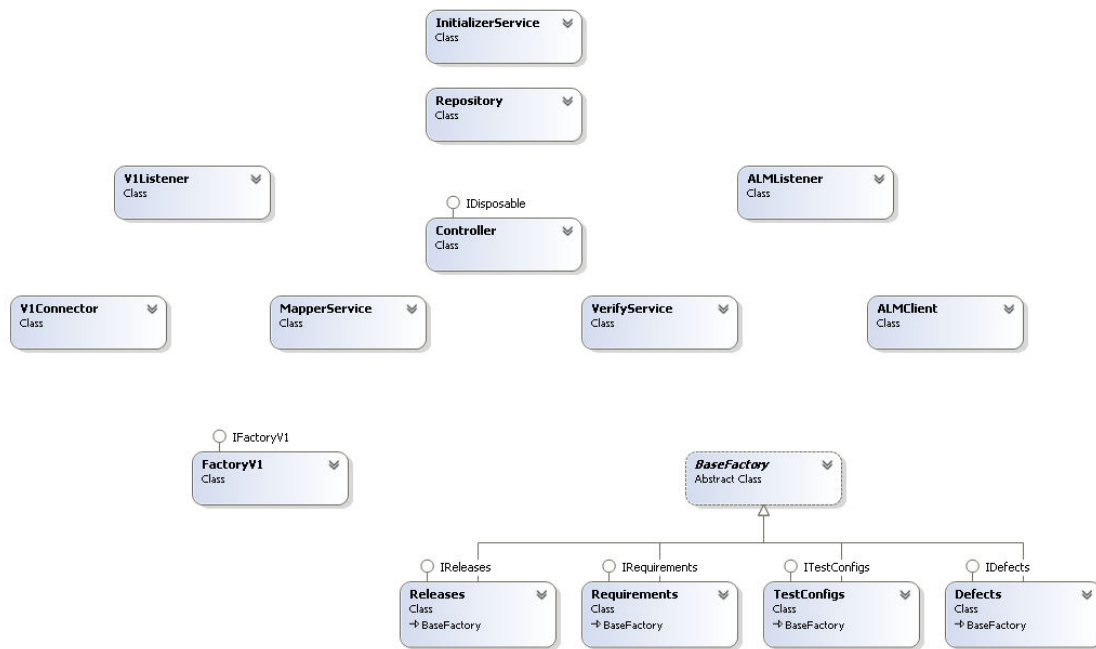


Figure 9 – The components of Synchronizer Core module

As shown at figure above the core consists of lot of smaller components. When I was designing the architecture I tried to keep to the rule of single responsibility to create easy maintainable and scalable software.

The closer look to each component from core will be in the next sections but for overall imagination of the process, the functionality of each component will be briefly mentioned here.

Before own synchronization process, the project needs to be scanned, initialized and missing entities loaded, that is the functionality of **InitializerService**, as initializing service works and entities are continuously created, those links are stored inside the **Repository**. **Controller** manages whole synchronization process, based on events created on one of the

**Listeners (V1 Listener, ALM Listener).** **VerifyService** checks if entity which should be synchronized contains all the data required for successful synchronization. **MapperService** works as a “bridge” between an entities, it converts ALM entity to V1 entity and vice versa. **Factories** are responsible for creating entities.

Application needs to be developed as multi-threaded, each listener needs its own thread to run properly also controller and services related to controller needs to run on separated thread to be able to catch the events from the listener.

From that point of view we will need three thread for one synchronization instance just for core. One thread will be needed for UI which is not depending on number of synchronization instances.

## 5.3 REST Client

### 5.3.1 Version One REST Client

Version One REST Client should help us to make communication between synchronizer and VersionOne. As I mentioned at limitations and bottlenecks section the REST Client for VersionOne exists but does not fit to synchronizer purpose because missing SSO and OAuth2 authentication and authorization support.

Even with those missing parts I figured out that will be much easier for me to extend the existing client then create whole client on my own and “reinvent wheel”.

REST Client for Version One is built on existing WebClient class inside .NET with some modifications. First of all I need to extend functionality of WebClient to be able upload OAuth2 string to Version One server. This functionality was achieved by extension method **UploadStringOAuth2** on figure 10.

```
public static class WebClientExtensions {
    public static string UploadStringOAuth2(this WebClient client,
        IStorage storage, string scopes, string path, string queryBody) {
        var creds = storage.GetCredentials();
        client.AddBearer(creds);
        try {
            return client.UploadString(path, queryBody);
        }
        catch (WebException ex) {
            if (ex.Status == WebExceptionStatus.ProtocolError) {
                if (((HttpWebResponse)ex.Response).StatusCode != HttpStatusCode.Unauthorized)
                    throw;
                var secrets = storage.GetSecrets();
                var authclient = new AuthClient(secrets, scopes, null, null);
                var newcreds = authclient.refreshAuthCode(creds);
                var storedcreds = storage.StoreCredentials(newcreds);
                client.AddBearer(storedcreds);
                return client.UploadString(path, queryBody);
            }
            throw;
        }
    }
}
```

Figure 10 – Extension method for uploading the OAuth2 string

Then I try to find out how to extend client to SSO login which takes me lot of time and research.

For authentication to Version One at CA Technologies is used **SiteMinder SAML 2.0 post binding protocol**. I need to make sure that extended REST client will be able to login through this protocol. For better understanding of SAML 2.0 protocol the description of the investigation of protocol will be mentioned here, it will also help us to understand the implementation.

**SAML 2.0 post binding protocol** (Security Assertion Markup Language) is XML-based protocol that uses security tokens containing assertions to pass information about end user between SAML authority – identity provider (Idp) and SAML customer – service provider (sp). On the figure bellow (figure 11) you can see scheme how SAML works.

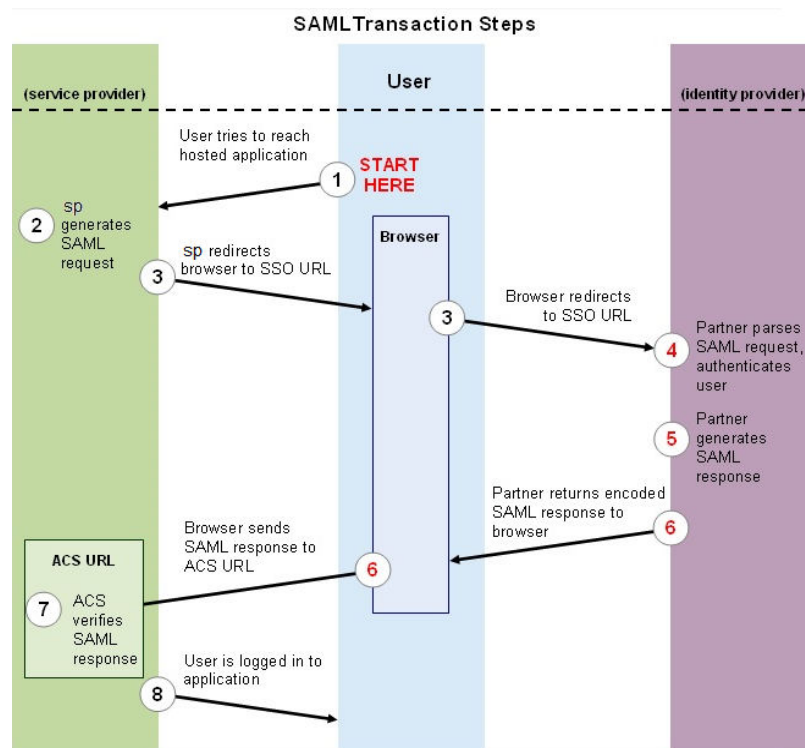


Figure 11 – Scheme of SAML transactions <sup>7</sup>

With usage of plugin for Firefox called HttpFox the transactions was traced down and the authorities for logging in was recognized as identify Service provider is **samlgwsn.ca.com** and Identity provider is **iwassosm.ca.com**.

<sup>7</sup> Source: <http://complispace.github.io/images/saml-transaction-steps.png>

Files created for modification of existing REST client can be found in **SynchronizerInstance.VersionOne.REST.SSOExtension** namespace. Those files are shown in figure below (Figure 12).

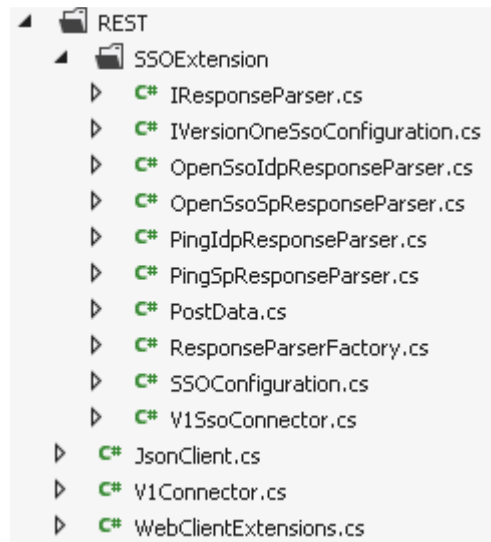


Figure 12 – Files for modification of existing REST Client

Most of work, as can be seen on the figure 4, was to write parsers that will handle responses and requests from service provider and identity provider.

For parsing the response I used combination of regular expressions and XPath. That depends on format of response if service is able to send response at XML format parsers works with XPath (Figure 13) otherwise regular expressions (Figure 14) are used.

```
private string SamlResponse { get { return GetValueFromNode("//input[@name=\"SAMLResponse\"]", "value"); } }  
private string RelayState { get { return GetValueFromNode("//input[@name=\"RelayState\"]", "value"); } }
```

Figure 13 – XPath to get SAMLResponse

```

private string SamlResponse {
    get {
        Regex samlR = new Regex("SAMLResponse\\\"value=\\\".*?\\\">");
        string samlResp = samlR.Match(code).ToString();
        samlResp = samlResp.Replace("SAMLResponse\\\"value=\\\"","");
        samlResp = samlResp.Replace("\\\">", "");
        return samlResp;
    }
}

```

Figure 14 – Regular expression to get SAMLResponse

To invoke SSO authentication instead of basic one is used **V1SsoConnector** class (Figure 15) used. The class implements **IAPICConnector**, which is interface of connector from existing REST client.

Newly created class V1SsoConnector handles only SSO login processes. Common login process used by VersionOne left to the existing REST client.

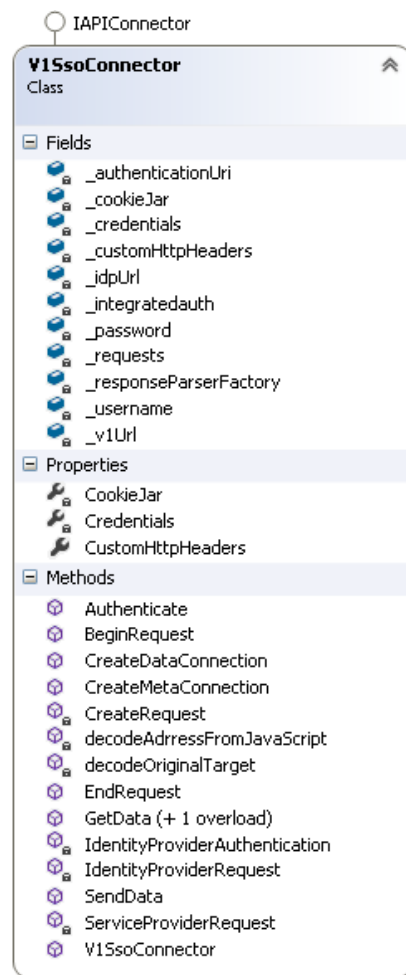


Figure 15 – Class diagram of V1SsoConnector



### 5.3.2 Application lifecycle management REST Client

REST API at ALM side is still under development so it does not offer as much functionality as needed. Because of the development status is client pretty easy. On the figure bellow (Figure 16) you can see main three classes of client - **RestConnector**, **ALMClient** and **Response**.

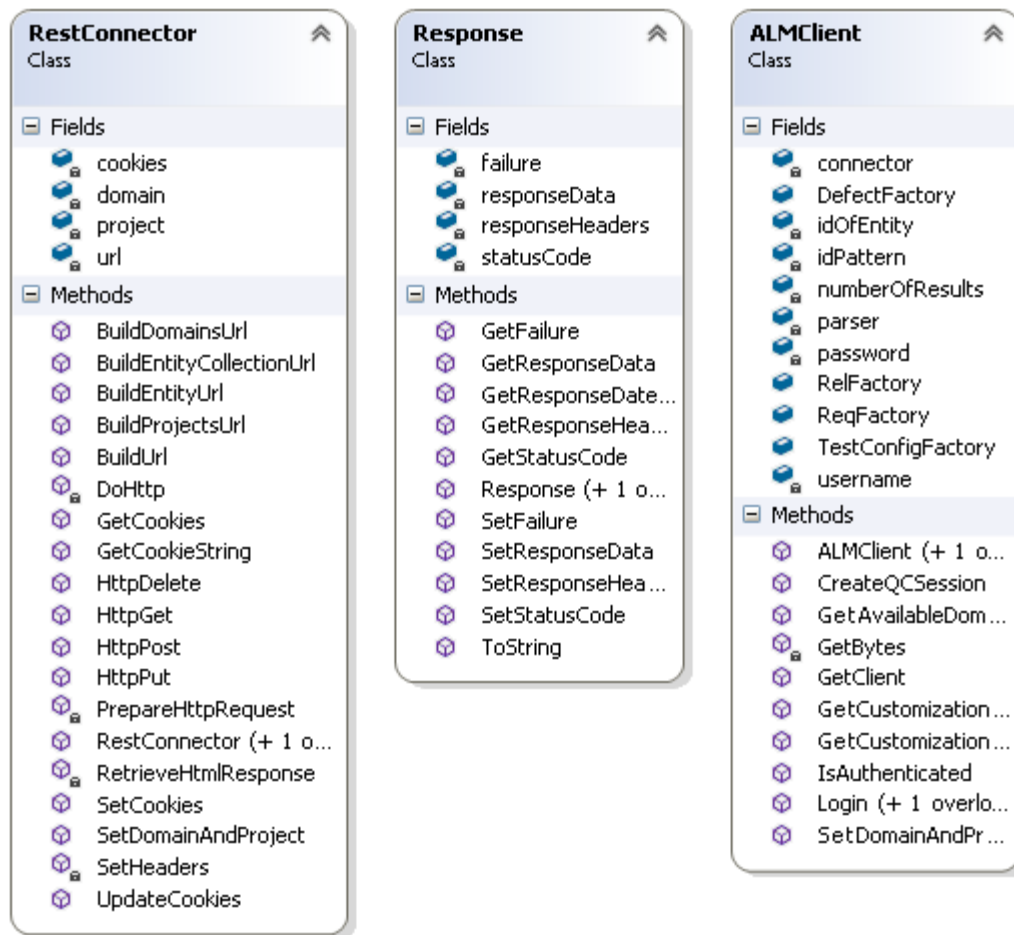


Figure 16 – Class diagram of main client components

**RestConnector** role is defines basic methods for HTTP requests - GET, POST, PUT, DELETE, saving cookies with login information and QCSession token. The most important method of RestConnector is **DoHttp** method Code snippet of the method is show on figure bellow (Figure 17).

```

private Response DoHttp(string type, string url,
                        string queryString, byte[] data,
                        Dictionary<string, string> headers, Dictionary<string, string> cookies)
{
    if ((queryString != null) && !queryString.Equals(""))
    {
        url += "?" + queryString;
    }
    HttpRequest con = (HttpRequest)WebRequest.Create(url);
    con.Method = type;

    Response ret;
    string cookieString = GetCookieString();
    try
    {
        PrepareHttpRequest(con, headers, data, cookieString);
        ret = RetrieveHtmlResponse(con);
        UpdateCookies(ret);
    }
    catch (WebException ex)
    {
        if (ex.Status == WebExceptionStatus.ConnectionClosed)
            throw new ConnectionClosedException("Connection unexpectedly closed!", ex, con);
        ret = RetrieveHtmlResponse(con);
    }

    return ret;
}

```

Figure 17 – DoHttp method

Based on the parameters you just specify **type** of request, and to which **URL** you sending request, if you are using filtering put filter query into **queryString**, parameter **data** holds data which are send to server, **header** and **cookies** are stored at Dictionary because consists of key and value.

On the figure bellow (Figure 18) you can see how this method is used for creating the GET request to server. GET request does not allow to send any data to server, so we set them as null.

```

public Response HttpGet(string url, string queryString, Dictionary<string, string> headers)
{
    try
    {
        return DoHttp("GET", url, queryString, null, headers, cookies);
    }
    catch (ConnectionClosedException ex)
    {
        throw ex;
    }
}

```

Figure 18 – DoHttp method for GET request

On the figure above function returns **Response** object which is just an object representation of HTTP response from ALM and it is used for easier handling of responses. Through the object can synchronizer easily extract information as response body, header, status code and also failure if request fails.

**AlmClient** connects request from RestConnector to functional blocks, through that client so you are able to login, logout, checks if user is authenticated.

For authentication to ALM is used basic authentication. Authentication is done basically through HTTP call GET with Authorization token. Authorization token should look like: "Basic base64encoded(username:password)". The example of authorization token used for basic authentication:

*Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==*

The login functionality is shown at the figure below (Figure 19).

```
public bool Login()
{
    String authenticationPoint = this.IsAuthenticated();
    if (authenticationPoint != null)
    {
        return this.Login(authenticationPoint, username, password);
    }
    CreateQCSession();
    return true;
}

private bool Login(string loginUrl, string username, string password)
{
    string creds = (username + ":" + password);
    string credEncodedString = Convert.ToBase64String(Encoding.UTF8.GetBytes(creds));

    Dictionary<string, string> map = new Dictionary<string, string>();
    map.Add("Authorization", "Basic " + credEncodedString);

    Response response = connector.HttpGet(loginUrl, null, map);

    bool ret = (response.GetStatusCode() == (int)HttpStatusCode.OK ? true : false);
    CreateQCSession();
    return ret;
}
```

Figure 19 – ALM Client login functionality

There are two functions for login. Public function checks if user is authenticated and returns null or URL of authentication point based on authentication state of user. If URL of authentication endpoint is returned the second function which is private, it is not visible out of class, create authentication token and send it to given authentication point.

At the end CreateQCSession is invoked to create QCSession token which is at ALM 11 returned automatically but at newer ALM versions you will need to create that token on your own. So as CA Technologies migrate to ALM 12 the synchronizer will still be able to reach REST ALM endpoint.

For creating, reading, updating and deleting operations at ALM are used factories. Factories differs just in few things so all factories are inherited from the **BaseFactory** where the available operations to entities are defined. Available factories and also methods you can see on class diagrams at figure 20. If synchronizer will be developed continuously new factories can comes up. Those are just minimal to proof of concept.

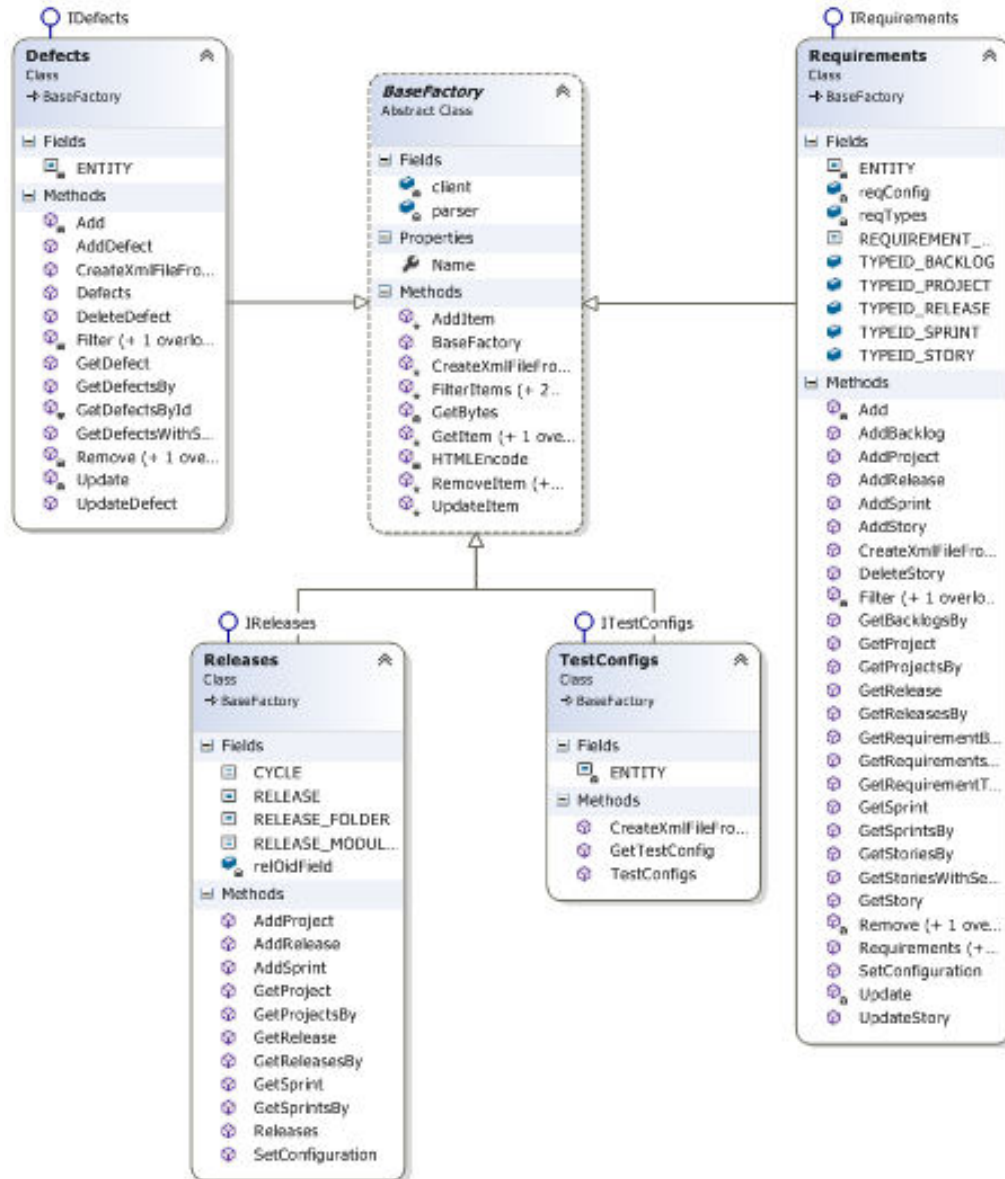


Figure 20 – Class diagram of available factories

BaseFactory returns responses from server in XML format. REST API offers two formats of response XML and JSON, this option is available through header Accept which needs to be set for each request, and the available values for Accept header are **application/xml** or **application/json**.

XML format was chosen for synchronizer because I'm more familiar with the XML processing then processing of JSON.

Example of response for GET request through ALM REST API is shown at figure bellow (Figure 21).

URL of the request:

***www.alm-dev.ca.com/qcbin/rest/projects/MAINFRAME/domains/AGILE/requirements/1***

```
▼<Entities TotalResults="1">
  ▼<Entity Type="requirement">
    ▼<Fields>
      ▼<Field Name="id">
        <Value>0</Value>
      </Field>
      ▼<Field Name="name">
        <Value>Requirements</Value>
      </Field>
      ▼<Field Name="has-linkage">
        <Value>N</Value>
      </Field>
      ▼<Field Name="alert-data">
        <Value/>
      </Field>
      ▼<Field Name="cover-count">
        <Value>0</Value>
      </Field>
      ▼<Field Name="has-traced-from">
        <Value>N</Value>
      </Field>
      ▼<Field Name="has-traced-to">
        <Value>N</Value>
      </Field>
      ▼<Field Name="father-name">
        <Value/>
      </Field>
    </Fields>
  </Entity>
</Entities>
```

Figure 21 – XML returned to GET request

XML is converted to object by class called **ResponseParser**. Class diagram is shown on figure 22.

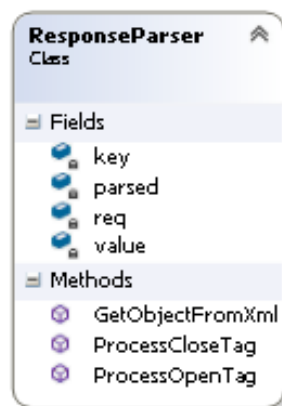


Figure 22 – Class diagram ResponseParser

**ResponseParser** extends existing `XmlReader` class available at .NET. Reader handles the given XML file from top to bottom and trigger defined events. The events are defined by user, mostly are events hooked on start and end elements of the XML file.

Function which starts whole parsing process is called **GetObjectFromXml** and the body of function is shown below on figure 23.

```
public List<AlmObject> GetObjectFromXml(string xml)
{
    parsed = new List<AlmObject>();
    XmlReader reader = XmlReader.Create(new StringReader(xml));

    while (reader.Read())
    {
        switch (reader.NodeType)
        {
            case XmlNodeType.Element:
                ProcessOpenTag(reader);
                break;
            case XmlNodeType.EndElement:
                ProcessCloseTag(reader);
                break;
            case XmlNodeType.Text:
                value = reader.Value;
                break;
        }
    }
    return parsed;
}
```

Figure 23 – Body of function `GetObjectFromXml`

Through switch is managed whole parsing process. In function **ProcessOpenTag** (Figure 24) are handled opening elements and **ProcessCloseTag** handles the ending elements (Figure 24) of XML file.

```
public void ProcessOpenTag(XmlReader reader)
{
    switch (reader.Name)
    {
        case "Entity":
            req = new AlmObject();
            break;
        case "Field":
            key = reader.GetAttribute("Name");
            break;
    }
}

public void ProcessCloseTag(XmlReader reader)
{
    switch (reader.Name)
    {
        case "Entity":
            parsed.Add(req);
            req = null;
            key = null;
            value = null;
            break;
        case "Field":
            req.GetFields().Add(key, value);
            value = "";
            break;
        case "":
            value = reader.Value;
            break;
    }
}
```

Figure 24 – `ProcessOpenTag` function on the left side and `ProcessCloseTag` on the right side

## 5.4 Factories

Factories allow to synchronizer or user, if he would like to use ALM REST Client to his own purpose, do CRUD operations over entities without deep knowledge of HP ALM REST API. As was mentioned before the number of factories is not final and it is just a minimum to proof of concept that synchronization can be done.

From available factories we have **Release**, **Requirement**, **TestConfig** and **Defect** factory. The division of factories is based on REST entity endpoint and it also keeps the same structure as modules inside HP ALM UI.

As the most visible part of synchronization takes place at Requirements and Defect module, those factories will be described.

### 5.4.1 Requirement factory

The **Requirements** is the name of factory for entities at requirement module. The requirement entities are specific because of the fact that they can be customized by user but from the REST API endpoint perspective are stored at one endpoint. The type is specified by **type-id** attribute of entity.

From the customization option of requirement types comes problems that factory needs to know all available type-id of entities used for synchronization. This information comes from configuration file and is stored in factory by SetConfiguration method (Figure 25).

```
public void SetConfiguration(Field reqConfig, Dictionary<string, ReqType> reqTypes)
{
    this.reqConfig = reqConfig;
    this.reqTypes = reqTypes;
    this.TYPEID_PROJECT = reqTypes.FirstOrDefault(x => x.Key == "Project").Value.Id;
    this.TYPEID_RELEASE = reqTypes.FirstOrDefault(x => x.Key == "Release").Value.Id;
    this.TYPEID_SPRINT = reqTypes.FirstOrDefault(x => x.Key == "Sprint").Value.Id;
    this.TYPEID_STORY = reqTypes.FirstOrDefault(x => x.Key == "Story").Value.Id;
    this.TYPEID_BACKLOG = reqTypes.FirstOrDefault(x => x.Key == "Backlog").Value.Id;
}
```

Figure 25 – SetConfiguration method of Requirement factory



As you can see method have two input parameters. The first parameter is **reqConfig** which specifies field where VersionOne ID will be stored inside ALM and the second one is called **reqTypes** and specifies type-id of entities like Project, Release etc. The values are stored at dictionary and LINQ expressions are used to search right requirement type-id.

The FirstOrDefault method is used for search of record x where key of record is named as Project. Same for other type-ids with different names. The Dictionary with reqTypes is created by configuration wizard and will be described in next section.

The methods for CRUD operations are created separately for each entity mainly for better readability and usability of client as standalone module.

For better imagination of factory functions the AddStory function will be shown on figure 26.

The **AddStory** function accepts AlmObject as input parameter and specifies the type-id of the requirement entity and pass AlmObject to more generic function Add.

The Add function converts AlmObject to XML and passes the XML with name of entity (requirement) to **AddItem** function (Figure 27) which is part of **BaseFactory**.

```
public AlmObject AddStory(AlmObject story)
{
    story.TypeID = TYPEID_STORY;
    return Add(story);
}

private AlmObject Add(AlmObject reqObject)
{
    string xml = CreateXmlFileFromObject(reqObject);
    return AddItem(ENTITY, xml);
}
```

Figure 26 – AddStory and Add function of requirement factory

```
protected AlmObject AddItem(string entity, string xml)
{
    string collectionUrl = client.BuildEntityCollectionUrl(entity);
    Dictionary<string, string> requestHeaders = new Dictionary<string, string>();
    requestHeaders.Add("Content-Type", "application/xml");
    requestHeaders.Add("Accept", "application/xml");
    Response response = client.HttpPost(collectionUrl, GetBytes(xml), requestHeaders);
    return parser.GetObjectFromXml(response.GetResponseDateAsString())[0];
}
```

Figure 27 – AddItem function

The **AddItem** function accepts as first input parameter the name of the entity and as second parameter the string at XML format which represents body of request (entity information in our case). Client builds URL of REST entity endpoint based on input parameter. The URL is built by client so information about the domain and project, where the synchronizer is hooked up, are populated to the request. Headers with all mandatory information like content-type and authorization token are also populated from client. For creation of new entity through REST is standardized to use the HTTP POST call so method `HttpPost` is used and sends request to server.

On the successfully created entity server returns RC 201 and in the body of response is XML file with the entity information. When creation was not successful server returns RC 40x or 500. The overview of possible RC is shown at figure 28. In case of RC 500, which stands for internal error, the body contains the XML file with specification off error.

Code	Cause
200	successful operations
201	successful POST operations that create a new entity
401	unauthenticated request
403	unauthorized operations
404	resource not found
405	method not supported by resource
406	unsupported ACCEPT type
415	unsupported request content type
500	Internal server error

Figure 28 – HP ALM REST return codes

### 5.4.2 Defect factory

The **Defects** is the name of factory for entities at defect module. The difference from **Requirements** is that defects module does not offer any option to modify the types of the

defects. To maintain consistency of factories, defects factory also implement AddDefect and Add methods (Figure 29).

As defect entity does not allow user to modify type **AddDefect** method just call Add method where is AlmObject transformed into XML and passed to **AddItem** function which is shown above at section about requirement factory (Figure 27).

```
public AlmObject AddDefect(AlmObject defect)
{
    return Add(defect);
}

private AlmObject Add(AlmObject reqObject)
{
    string xml = CreateXmlFileFromObject(reqObject);
    return AddItem(ENTITY, xml);
}
```

Figure 29 – AddDefect and Add method of defect factory

## 5.5 Synchronizer configuration

To make synchronizer process customizable. There needs to be some configuration file which will hold the customization data and also synchronizer should be able to load and save the data so user does not need to create new synchronizer configuration after restart or reboot machine also needs to be able to store many configurations not just one.

As there is a lot of information which user needs to define and also there is much room to make mistake in creation of configuration file. That is the reason why configuration wizard was created. It guides user through the whole customization process and at the end will generate file with configuration data.

File is at XML format and user can create as many configurations as he wants to, only restriction is that configurations must have unique instance name.

Configuration wizard is embedded into synchronizer software to opens it you just press **Add new instance** button at main page (Figure 30) and wizard will pops up. Configuration manager will be described from developer point of view. User point of view will be described at user guide. Which will be created as standalone.



Figure 30 – Main page of synchronizer

The customization process is divided into seven parts. The purpose of each wizard page will be described below and also if any interesting process runs at background I will described it a little. On many wizard pages are data extracted from tools site for user comfort.

### 5.5.1 General information

On the first page which is shown below (Figure 31) is general information about the instance name, credentials and also the URLs of both tools. Because synchronizer is developed mainly for CA Technologies internal usage and SSO login is used to access both tools the credentials can be placed to this page, because both tools accepts same username and password.

Instance name field is monitored so user is unable to set instance name which is already in use also URL fields are checked to be in right format.

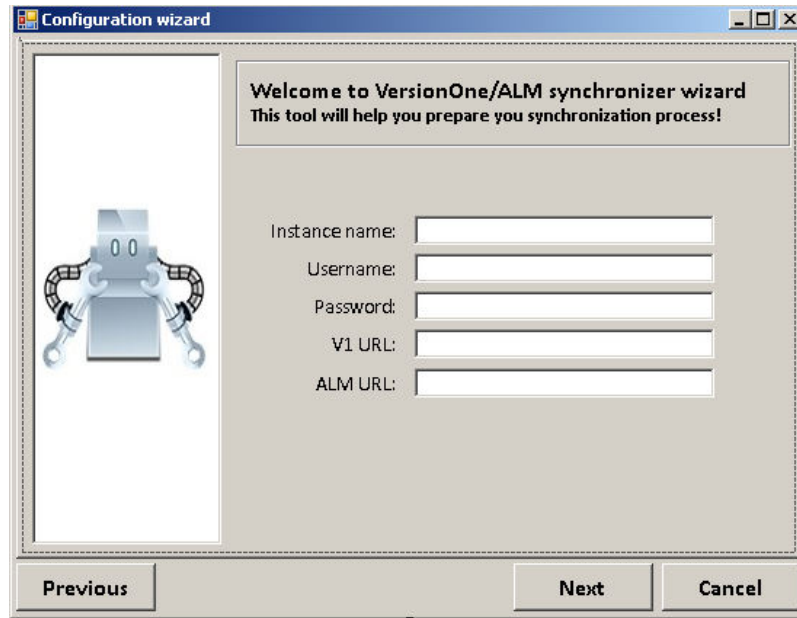


Figure 31 – General info page

## 5.5.2 OAuth2 settings

After general info is stored we will be moved to next page where wizard helps user to set up OAuth2 authentication against VersionOne.

For this page there are two possible scenarios, the first scenario appears as new configuration file is created, the second one appears when the configuration file is modified. Both possibilities are shown at figure below (Figure 32).

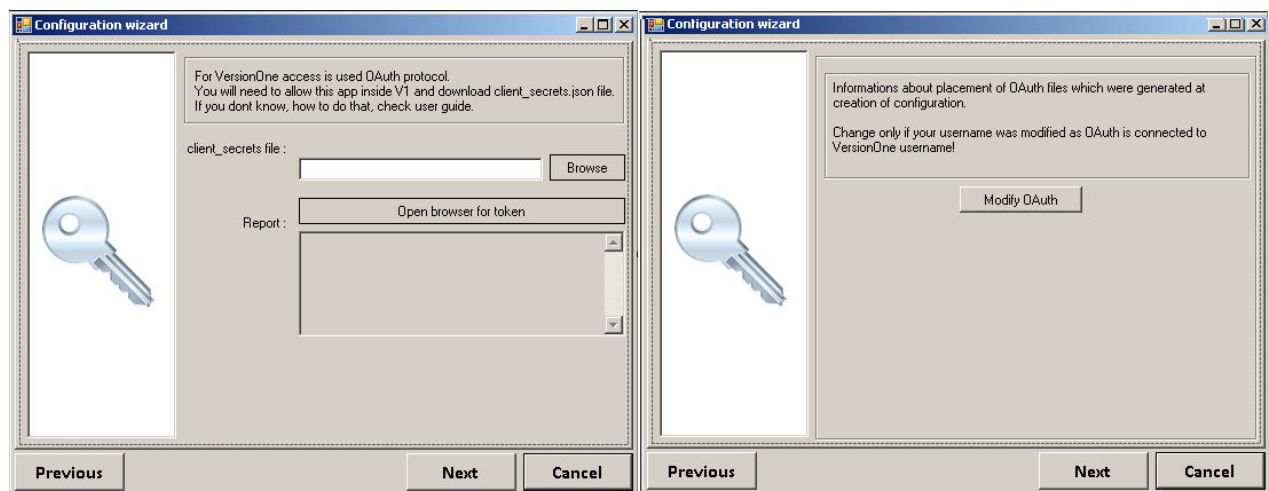


Figure 32 –Page for creation of new configuration is on the left side, for the modification on the right side

On the new creation page you need to put path to **client\_secrets.json** file which is generated by VersionOne.

The VersionOne does not allow to third-party programs connection to query.v1 endpoint until the program gets permission to access this endpoint by user. The permission consist by two files **client\_secrets.json** and **secret\_credentials.json**.

The OAuth2 settings page accepts the client\_secrets.json file and based on that generates the secret\_credentials.json file. This generation is done by GrantTool which is utility program from VersionOne developers to generating secret\_credentials.json file.

**GrantTool** accepts token which can be obtained through the URL which is composed from the information stored at client\_secrets.json file.

Wizard will do whole process programmatically so user just need to know path to the client\_secrets.json file and put this path into prepared text box or can locate it through the Browser button, which will invoke the file manager. As the path to the file is set “Open browser for token” button becomes available.

Two possible scenarios can occur after clicking the button, first one is, that everything runs without a problem and you will get noticed about success through the report box by message “Successfully saved credentials to stored\_credentials.json”, this situation can be seen on figure 33.

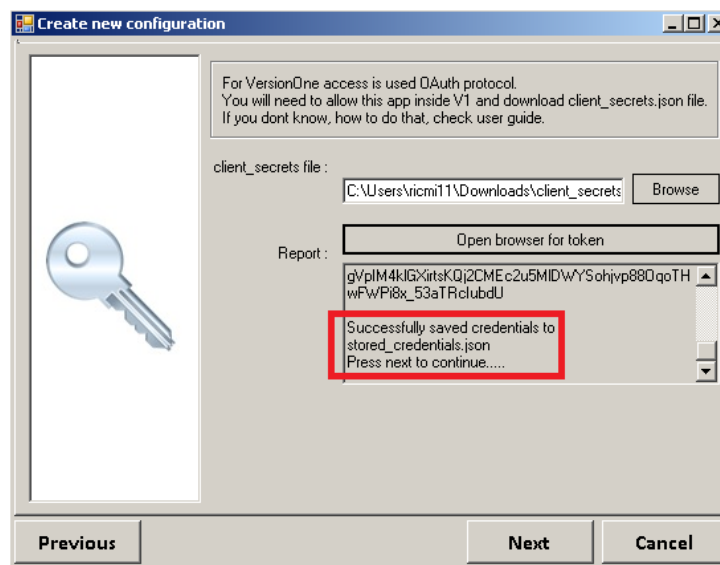


Figure 33 – Token was created successfully

The second scenario occurs if synchronizer is unable to locate elements on page which are used to process the first scenario. Browser with the page for token will be opened and user just needs to allow (click Allow button) the connection of synchronizer to VersionOne and copy-paste token from next page into input box which will pops up right after page is opened. The second scenario is shown on figure 34.

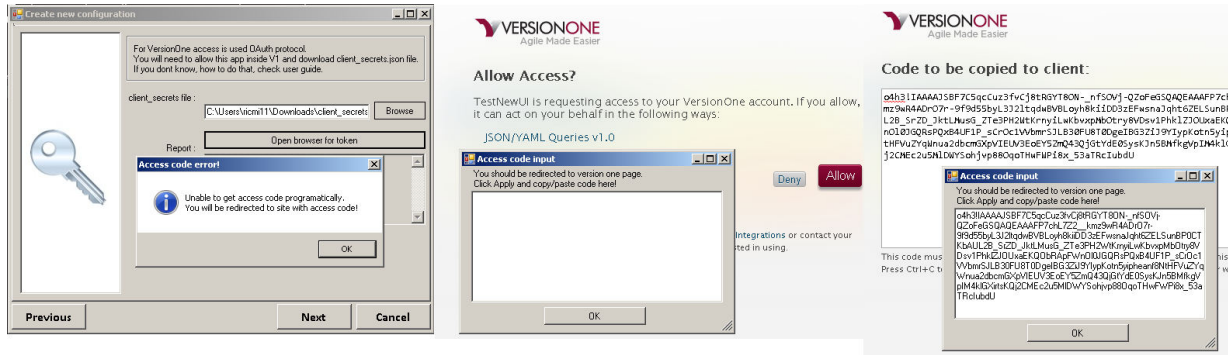


Figure 34 – Manually accessing the OAuth2 token

### 5.5.3 Project linkage

After the connection to VersionOne can be established we are able to choose which projects should be synchronized. The data about available projects for authorized user are extracted from the tools before the page is loaded.

Page is shown at figure 35 and allows to choose which project and release on VersionOne side will be synchronized to which domain and project on the side of ALM.

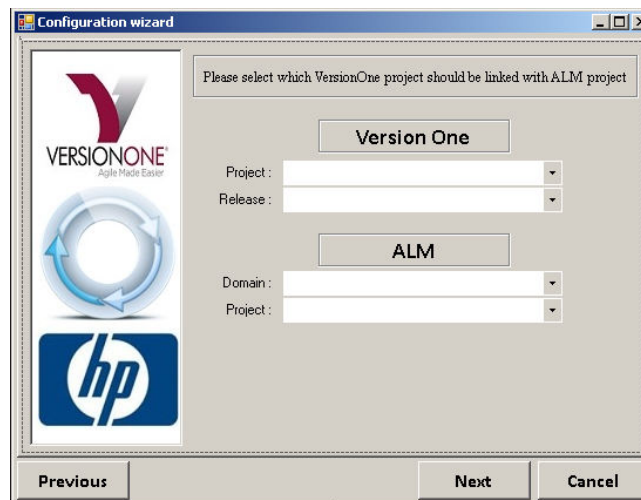


Figure 35 – Page with available projects

### 5.5.4 Entities customization

The next page of wizard is shown on figure 36 and user is able to set which entities should be synchronized - Stories, Defects or Stories and Defects, the representation entity of VersionOne release at ALM – Milestone, Cycle and create mapping between fields this option is there because the naming convention in both tools can differ.

Entities have predefined default fields (Name, Id and version stamp/time stamp) for synchronization, other fields can be added to synchronization process through “Configure” buttons. Configure buttons becomes available based on chosen entities for synchronization (e.g. if I choose as synchronize entity just stories configure button for defect fields mapping will not be available).

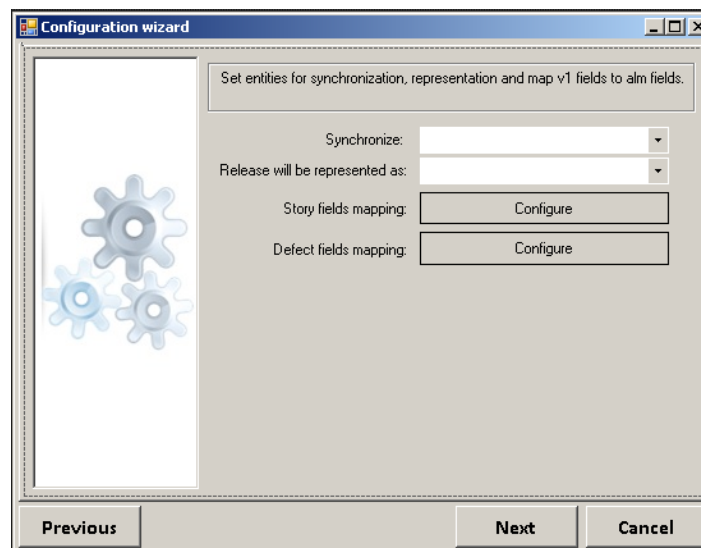


Figure 36 – Entity customization

Configure button opens form (Figure 37) with overview about mapped fields (default fields does not appear).

Available actions for mapping are Add or Remove, for Add button next form pops up (Figure 38) which allows us to map V1 fields to ALM field. The available fields are extracted from each tool based on selected projects in previous steps, because each project can have customization which suits best to the project.



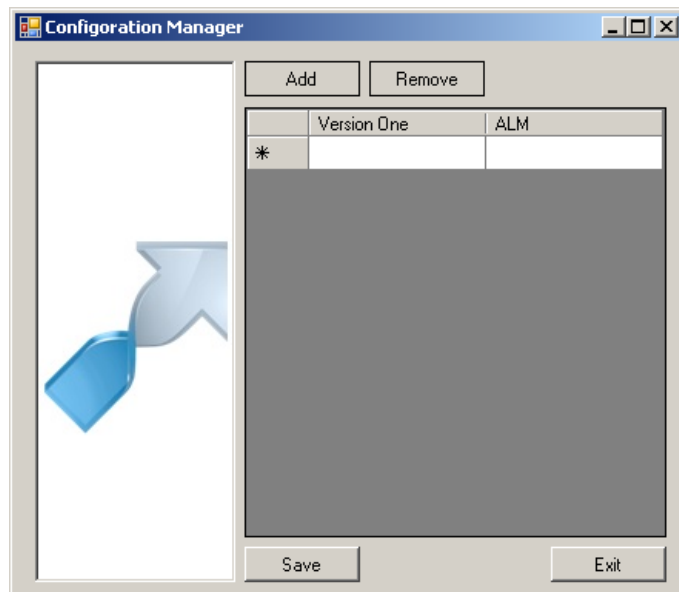


Figure 37 – Overview of mapped fields

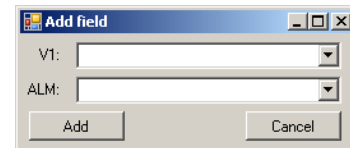


Figure 38 – Add field form

To save created mapping just click save button at overview form (figure 37) and mapping will be added to configuration file. Save button also invokes action which extracts all data about chosen fields.

Fields can have unique type, can be required or can be relation. Relation fields are special to Version One and are represented by Value and ID. To make sure that synchronizer will be able to “understand” to relation data. The information about relations needs to be extracted from tools.

Example of field - **status which is represented as relation** (Table 1)

Statuses of Status field	
Value	ID
In Progress	StoryStatus:134
Done	StoryStatus:135
Accepted	StoryStatus:137
Ready for Test	StoryStatus:3913

Table 1 – Status relation field

### 5.5.5 IDs and Requirements mapping

At top of the page (Figure 39) user specifies to which field in ALM will be the value of VersionOne ID inserted. This option is allowed to user because there is lot of teams at CA Technologies and every team use their own customization and also fields.

At the bottom of the page (Figure 39) user set how the entities from VersionOne (Project, Release, Sprint etc.) will be represented inside the ALM requirement module. This option is there because whole requirement module can be customized and also entities of that module can be customized.

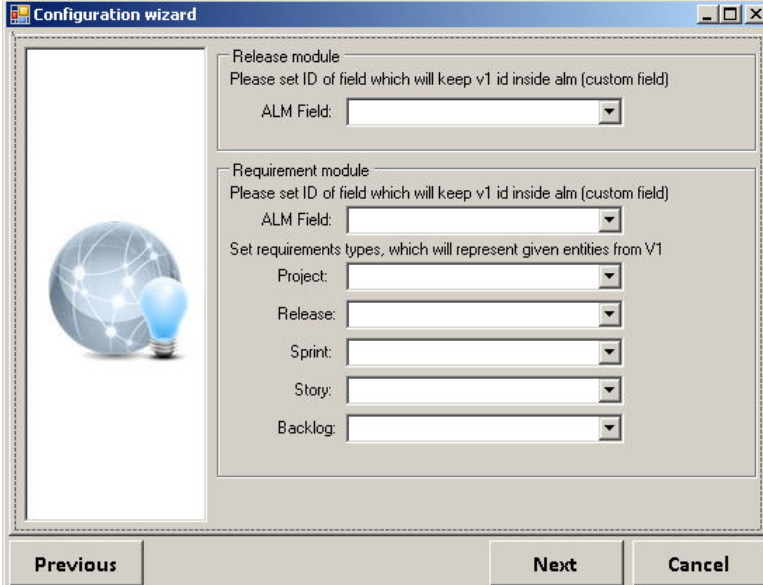
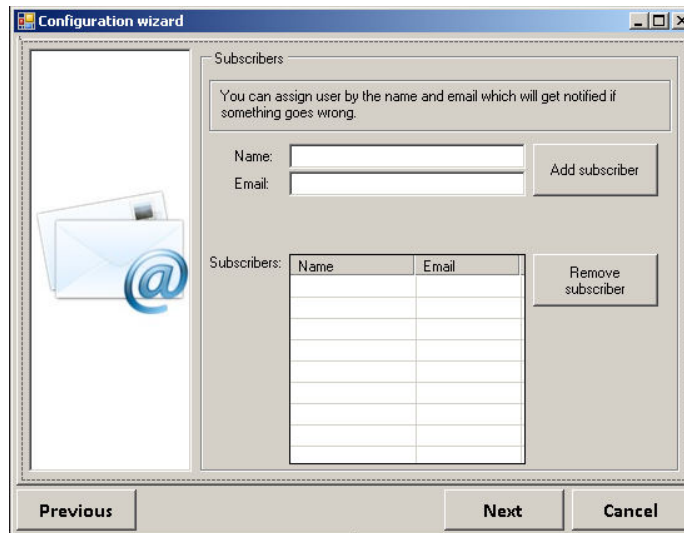
The image shows a 'Configuration wizard' dialog box. On the left is a graphic of a globe with a lightbulb. The right side contains two sections: 'Release module' and 'Requirement module'. Each section has a label 'Please set ID of field which will keep v1 id inside alm (custom field)' followed by an 'ALM Field:' dropdown menu. The 'Requirement module' section also includes a label 'Set requirements types, which will represent given entities from V1' followed by five dropdown menus for 'Project', 'Release', 'Sprint', 'Story', and 'Backlog'. At the bottom are 'Previous', 'Next', and 'Cancel' buttons.

Figure 39 – Form for ID and requirement mapping

### 5.5.6 Subscribers

Subscriber page (Figure 40) allows user to subscribe users for notifications if something goes wrong with given instance of synchronization, as synchronizer is designed as server application and should run on server machine.

The subscriber consists of Name and Email address where notifications will be delivered, those information are used by MailService at Synchronizer Core module.



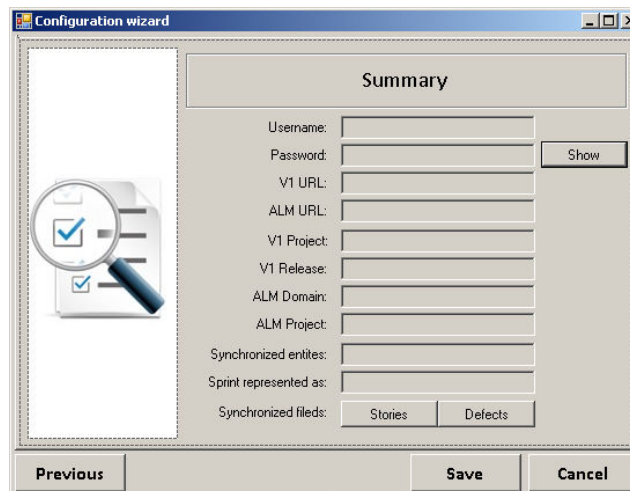
The screenshot shows a 'Configuration wizard' window with a 'Subscribers' tab. On the left is an icon of an envelope with an '@' symbol. The main area contains a text box stating: 'You can assign user by the name and email which will get notified if something goes wrong.' Below this are input fields for 'Name:' and 'Email:', followed by an 'Add subscriber' button. A table with two columns, 'Name' and 'Email', is present, with several empty rows. To the right of the table is a 'Remove subscriber' button. At the bottom are 'Previous', 'Next', and 'Cancel' buttons.

Figure 40 – Form for adding of subscribers

### 5.5.7 Summarization of the configuration

Last page (Figure 41) just summarizes the information about configuration created in previous steps. This page is there mainly for user to see all information at one place.

The next button from previous pages was changed to save button which will generate the **SyncConfiguration object** and also stores the configuration at local drive as XML file.



The screenshot shows a 'Configuration wizard' window with a 'Summary' tab. On the left is an icon of a magnifying glass over a document with checkmarks. The main area lists configuration details: 'Username:', 'Password:', 'V1 URL:', 'ALM URL:', 'V1 Project:', 'V1 Release:', 'ALM Domain:', 'ALM Project:', 'Synchronized entites:', 'Sprint represented as:', and 'Synchronized files:'. Each label is followed by an input field. A 'Show' button is next to the 'Password:' field. At the bottom of the 'Synchronized files:' section are two buttons: 'Stories' and 'Defects'. At the very bottom are 'Previous', 'Save', and 'Cancel' buttons.

Figure 41 – Summarize of information

### 5.5.8 Read/Write of configuration file

As was mentioned configuration is saved to XML file but inside the synchronizer is used SyncConfiguration class for easier manipulation with customization/configuration information. Class diagram is shown below on figure 42.

For creating XML file from SyncConfiguration class is used **XMLWriter** and for creating SyncConfiguration object from XML file is used **XMLReader**. Class diagrams can be seen on figure 42.

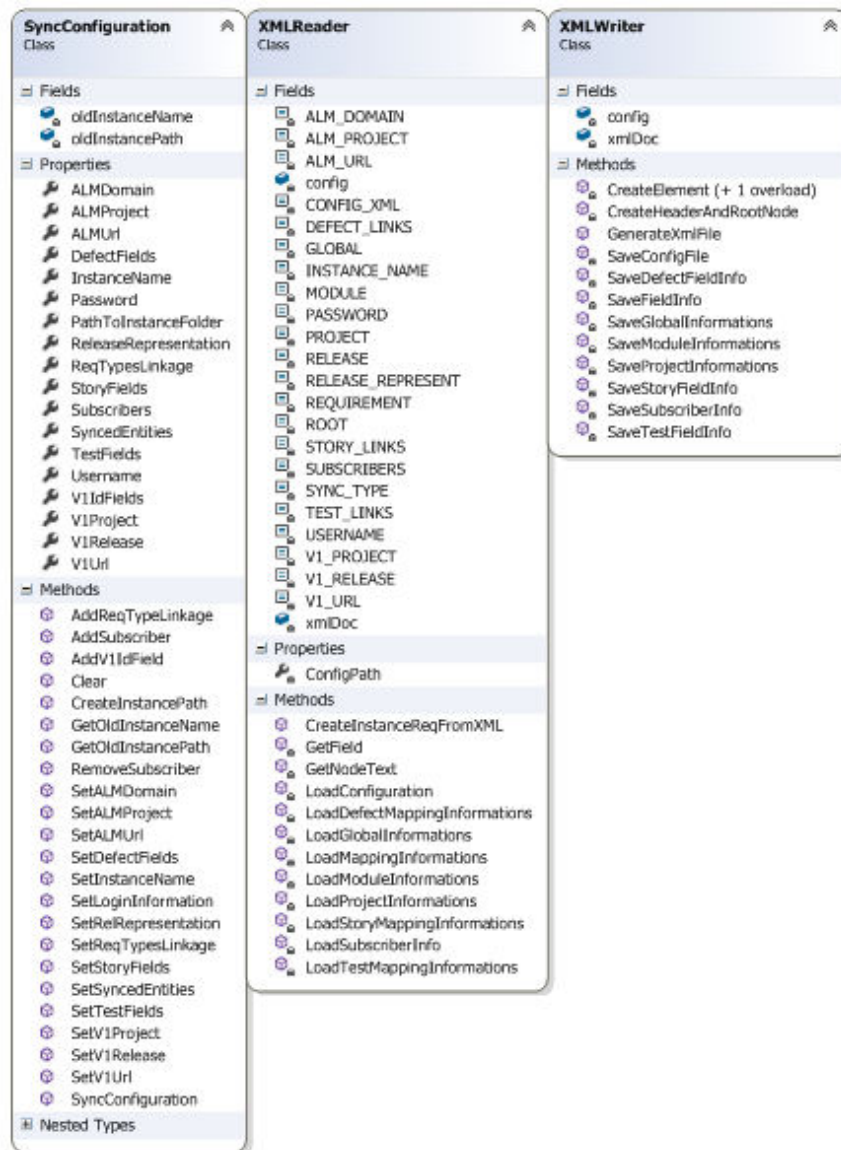


Figure 42 – Class diagrams

Right – XMLWriter | Middle – XMLReader | Left – SyncConfiguration

### 5.5.9 Password encryption/decryption manager

Because XML file is stored at local drive and contains user-sensitive data like password it needs to be encrypted before it is saved to local drive and decrypted on loading of configuration file.

Class responsible for password encryption/decryption is called **PasswordManager**, class diagram is shown on figure 43.

**PasswordManager** consists only from two methods Decrypt and Encrypt. Code of methods will not be published from security reasons inside the thesis. Class is designed as wrapper class for RijndaelManaged class and also implements PasswordDeriveBits method both methods are part of .NET framework.

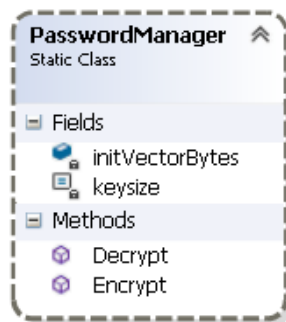


Figure 43 – Password manager class diagram

## 5.6 Synchronizer core

It is the module responsible for the synchronization process. The core needs to be configured by the configuration file created in previous section without the configuration file is unable to start synchronization. The configuration file is passed to the entry point of core as input parameter. Architecture of core is shown at section about architecture of whole synchronizer (Section 5.2).

In the next subsections will be each component of module described.

## 5.6.1 Initializer Service

Used for initialization part of synchronization. Service passes through all existing entities at projects which are currently synchronized and if it finds same entity in both tools checks the date of entity and take the data of entity with later timestamp otherwise the entity is created in VersionOne or ALM depends on tool where the entity comes from.

At the figure bellow (Figure 44) is described method for creating new entities.

```
private AlmObject CreateEntity<T>(TreeStructure.TYPE_OF_NODE type, T factory, V1Object item, Node parentNode)
{
    MethodInfo method = typeof(T).GetMethod("Add"+type.ToString());
    switch (type)
    {
        case TreeStructure.TYPE_OF_NODE.Project:
            return (AlmObject)method.Invoke(factory, new Object[] { item.Name });
        case TreeStructure.TYPE_OF_NODE.Release:
            return (AlmObject)method.Invoke(factory, new Object[] { item, parentNode.Alm_Id.ToString() });
        case TreeStructure.TYPE_OF_NODE.Sprint:
            return (AlmObject)method.Invoke(factory, new Object[] { item, parentNode.Alm_Id.ToString() });
        case TreeStructure.TYPE_OF_NODE.Backlog:
            return (AlmObject)method.Invoke(factory, new Object[] { parentNode.Alm_Id.ToString() });
        case TreeStructure.TYPE_OF_NODE.Story:
            return (AlmObject)method.Invoke(factory, new Object[] { ConvertToReqObject(item,parentNode) });
        case TreeStructure.TYPE_OF_NODE.Defect:
            return (AlmObject)method.Invoke(factory, new Object[] { ConvertToReqObject(item, parentNode) });
        default:
            return null;
    }
}
```

Figure 44 – Body of generic CreateEntity method

**CreateEntity** is generic method where generic parameter sets which factory will be used for creation of entity, The reason why there is more factory options is that entity on ALM side needs to be created at release module (Release factory) and also at requirement module (Requirement factory).

Input parameters of method are:

- **type** – type of entity which should be created (story, defect and etc.)
- **factory** – generic parameter stands for Requirements or Releases (ALM modules).
- **item** – data model of entity from V1, entity which needs to be created at ALM
- **parentNode** – parent node for created entity ( e. g. sprint node for story)

On the first line of code (GetMethod) gets the Add method for given entity type from factory. This way of referencing the method was chosen because of standardized naming convention.

The naming convention for factories looks like Add + name of entity (e. g. AddStory, AddDefect). Switch logic just make sure that method will be invoked with the right parameters.

As the result of initializer service process is same structure of projects at both tools, that structure can be seen on figure 45. The entities of project are at the same point, have same attributes etc.

Last step of initializer service is return collected information about entities to the repository. The returned information contains the V1 ID, V1 version stamp, ALM ID and ALM version stamp.

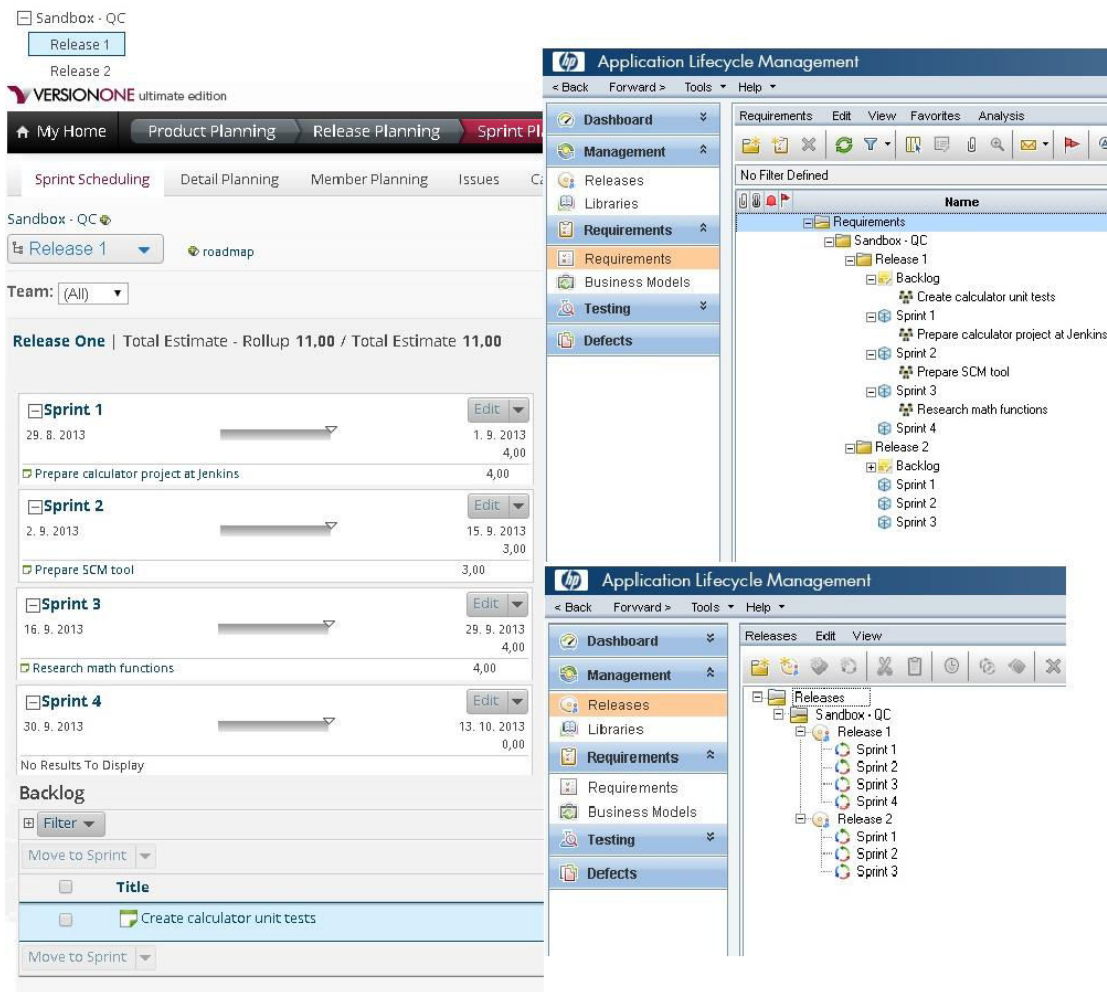


Figure 45 – Project structure after InitializerService run

Left side of the figure shows VersionOne project **Sandbox – QC** and it have **4 sprints** planned for **Release 1**. On top right side can be seen requirements module of ALM where is same structure created by InitializerService, all those entities are linked to release module (bottom right of the figure) where is created same structure but that structure ends up with sprints.

It is because the module is used basically for measuring of project KPIs for whole project, release or sprint.

## 5.6.2 V1 Listener

The V1 Listener component is responsible for monitoring the VersionOne and triggers defined events if they occurs at VersionOne.

Component is “listening” to Version One through REST at query.v1 endpoint. The comparison of available endpoints is shown below at table 2.

The requirements for listener are – fast, read-only. The query.v1 endpoint allow multiple queries and it is read-only endpoint so it suits better to listener purpose.

<b>query.v1</b>	<b>rest-1.v1</b>
Read-only	Read, write, and update
Multiple queries	Single queries
JSON serialization	XML serialization

**Table 2 – Available endpoints**

**V1 Listener** defines 5 events that are monitored, those events are shown at figure 46.

```
public event EventHandler<EntityId> V1ItemCreated;
public event EventHandler<EntityId> V1ItemDeleted;
public event EventHandler<EntityId> V1ItemUpdated;
public event EventHandler<ListenerInitialized> ListenerInitialized;
public event EventHandler<ExceptionEvent> V1ErrorOccured;
```

**Figure 46 – V1Listener events**



The first three events (**V1ItemCreated**, **V1ItemDeleted** and **V1ItemUpdated**) are self-explanatory, occurs if entity is created, deleted or updated. **ListenerInitialized** event is triggered as listener is initialized and ready to work. Last event **V1ErrorOccured** is thrown if something goes wrong (e.g. listener lost connection to VersionOne server).

Monitoring function for update event is called Update and it is shown on figure below (Figure 47).

```
private void Update(Constants.Entities entity, string filter, string by)
{
    try
    {
        string queryBody = @"
from: " + entity.GetV1Representation() + Environment.NewLine +
    @"select:
- Name
- ID
- Moment
filter:
" + by + "=" + filter + """;

        var resultSets = connector.JClient.GetResultSets(queryBody).ToArray();
        foreach (var result in resultSets[0])
        {
            string id = JSON.JsonParser.ParseValue(result["ID"].ToString());
            var existingItem = registeredItems.FirstOrDefault(x => x.Key == id);
            if (!existingItem.Equals(new KeyValuePair<string, string>()) &&
                existingItem.Value != result["Moment"].ToString())
            {
                registeredItems[existingItem.Key] = result["Moment"].ToString();
                OnItemUpdated(new EntityId(existingItem.Key, entity));
            }
        }
    }
    catch (Exception ex)
    {
        throw new Exception("GenericMethod:Update", ex);
    }
}
```

Figure 47 – Update function

In queryBody variable is defined YAML request to query.v1 endpoint. As YAML is human readable format, we can read the request like: From (**first parameter: entity**) select name, ID and moment for field where field with the name (**third parameter: by**) equals to the value (**second parameter: filter**).

As you may notice the entity variable is defined as enum but method GetV1Representation which is used is not definitely default enum method. The method is defined as an extension method which is available from .NET 3.5 framework.

Extension method **GetV1Representation** is used because V1 using names project, release and sprint just in UI and at backend are used special names and because of the fact that in whole synchronizer are entities defined like project, release etc. this extension method is used to convert synchronizer naming convention to V1 backend naming convention. Body of extension method can be seen on figure 48.

```
public static string GetV1Representation(this Constants.Entities entity)
{
    switch (entity.ToString())
    {
        case "Project":
            return "Scope";
        case "Release":
            return "Scope";
        case "Sprint":
            return "Timebox";
        default:
            return entity.ToString();
    }
}
```

Figure 48 – Definition of extension method GetV1Representation

Back to the figure 47 when **resultSets** is returned listener needs to resolve if changes were made. It is passes through all returned entities and checks if entity exist in repository and if exist checks if version number is same as in repository. If version number of returned entity differs from the one at repository, OnItemUpdate (Figure 49) is invoked with EntityId event arguments.

**EntityId** event extends EventArgs with properties of entity type (e.g. Story, Defect) and ID of the entity.

```
private void OnItemUpdated(EntityId e)
{
    EventHandler<EntityId> handler = V1ItemUpdated;
    if (handler != null)
    {
        V1ItemUpdated(this, e);
    }
}
```

Figure 49 – OnItemUpdated function

**OnItemUpdated** function just checks if event have defined any subscriber, if subscriber is defined the handler is not null and event is raised. In our case the only one subscriber for V1ItemUpdated event is Controller component.

### 5.6.3 ALM Listener

The ALM listener have same responsibility as V1Listener but against ALM server. Defines five events also and those events are shown at figure 50.

```
public event EventHandler<EntityId> ALMItemCreated;  
public event EventHandler<EntityId> ALMItemUpdated;  
public event EventHandler<EntityId> ALMItemDeleted;  
public event EventHandler<ExceptionEvent> ALMErrorOccured;  
public event EventHandler<ListenerInitialized> ListenerInitialized;
```

Figure 50 – ALMListener events

ALMListner class is almost same as V1Listener, but few requests need to be implemented in another way.

ALM offers only one REST endpoint so ALM listener is connected to endpoint called **rest**. This endpoint is a bit similar to rest-1.v1 endpoint on VersionOne side, rest-1.v1 specifications are introduced above at subsection about V1Listener (Subsection 5.6.2 - Table 2).

The biggest problem is that endpoint supports only single query, so synchronizer needs to run on machine with fast internet connection and good connectivity to both applications. With high response times is reliability falling down.

From single query ability another problem comes, we do not want to get responses from listener with redundant data. It is the reason why requests from listener are as much specific as can be. For that purpose is filtering done on server side through special query requests which secures that we do not transfer redundant data. Data are filtered based on hierarchical path and modification timestamp.

Requests created as is described above are able to resolve only new or update event. For delete event needs to be send another request, this request gets number of entities under the project and ID of the entities. Based on the comparison of returned list of IDs and list of IDs stored at repository is synchronizer able to recognized deleted entities and trigger delete event.

### 5.6.4 Controller

It is the “a brain” of synchronization. As synchronization starts, controller accepts configuration file which is represented by SyncConfiguration object.

Based on information from the configuration file controller instantiate all components needed for synchronization and specify the behavior of those components based on the data from configuration file and then starts the process of synchronization.

This phase is called the initialization phase and it is shown at figure 51 just to make some overview about what services are initialized before the process of synchronization is started.

```
private void Initialize()
{
    InitALMClient();
    ConnectToALM();
    InitV1Client();
    ConnectToV1();
    InitRepository();
    InitMapperService();
    InitFactories();
    InitInitService();
    InitVerifyService();
    InitMailService();
}
```

Figure 51 – The Initialization phase

As an example how the initialization of any service look like was chosen verify service initialization (InitVerifyService). The body of the initialization of the function is shown at figure 52. From the figure we can see that first is extracted info about customization (required fields at projects) and then service is initialized.

```

private void InitVerifyService()
{
    // Extracting the required ALM fields from configuration object
    var requiredAlmFields = new Dictionary<string, Dictionary<Field, Field>>();
    requiredAlmFields.Add(Constants.Entities.Story.ToString(),
        config.StoryFields.Where(x => x.Value.IsRequired)
            .ToDictionary(x => x.Key, y => y.Value));
    requiredAlmFields.Add(Constants.Entities.Defect.ToString(),
        config.DefectFields.Where(x => x.Value.IsRequired)
            .ToDictionary(x => x.Key, y => y.Value));

    // Extracting the required V1 fields from configuration object
    var requiredV1Fields = new Dictionary<string, Dictionary<Field, Field>>();
    requiredV1Fields.Add(Constants.Entities.Story.ToString(),
        config.StoryFields.Where(x => x.Key.IsRequired)
            .ToDictionary(x => x.Key, y => y.Value));
    requiredV1Fields.Add(Constants.Entities.Defect.ToString(),
        config.DefectFields.Where(x => x.Key.IsRequired)
            .ToDictionary(x => x.Key, y => y.Value));

    // Initializyation of verify service with defined required fields
    verifyService = new VerifyService(requiredAlmFields, requiredV1Fields);
}

```

Figure 52 – Initialization of Verify service

After the services are initialized the synchronization process can start (Figure 53).

```

public void Start()
{
    Initialize();

    initService.CreateMissingEntities();
    mapperService.SetInformationsAboutStructure(initService.RequirementsMap, initService.ReleasesMap);

    RegisterV1Events(initService.RequirementsMap);
    RegisterALMEvents(initService.RequirementsMap, initService.ReleasesMap);
    StartListeners();

    Unlock(createDriver);
    Unlock(updateDriver);
    Unlock(deleteDriver);
}

```

Figure 53 – Start of synchronization process

First of all we need to prepare the projects for synchronization, the entities needs to be unified. The unification process is done by InitializerService (Subsection 5.6.1).

As the projects are unified we can pass information about created structure to MapperService.

Then is time to subscribe events from listeners. Subscription of events is shown on figure 54. The functions **RegisterV1Events** and **RegisterALMEvent** are responsible for creation of listeners and also subscription of controller methods for listener events.

From the picture you can easily see what event argument is passed with event and which method will be invoked for given event and also you can see that create, update and delete events from listener invokes different functions from controller but events **ListenerInitialized** and **ALMErrorOccured** and **V1ErrorOccured** points to same method.

```
public void RegisterV1Events(TreeStructure RequirementsMap)
{
    v1listener = new V1Listener(versionOne, config);
    v1listener.V1ItemCreated += new EventHandler<EntityId>(V1Listener_Created);
    v1listener.V1ItemUpdated += new EventHandler<EntityId>(V1Listener_Updated);
    v1listener.V1ItemDeleted += new EventHandler<EntityId>(V1Listener_Deleted);
    v1listener.V1ErrorOccured += new EventHandler<ExceptionEvent>(Listener_Exception);
    v1listener.ListenerInitialized += new EventHandler<ListenerInitialized>(ListenerUp);
    v1listener.RegisterItems(RequirementsMap);
}

public void RegisterALMEvents(TreeStructure RequirementsMap, TreeStructure ReleasesMap)
{
    almListener = new ALMListener(alm);
    almListener.ALMIItemCreated += new EventHandler<EntityId>(ALMListener_Created);
    almListener.ALMIItemUpdated += new EventHandler<EntityId>(ALMListener_Updated);
    almListener.ALMIItemDeleted += new EventHandler<EntityId>(ALMListener_Deleted);
    almListener.ALMErrorOccured += new EventHandler<ExceptionEvent>(Listener_Exception);
    almListener.ListenerInitialized += new EventHandler<ListenerInitialized>(ListenerUp);
    almListener.RegisterItems(RequirementsMap.GetNodes(), RequirementsMap.Root.HierarchicalPath, ReleasesMap);
}
```

Figure 54 – Creation of listeners and subscription to events from them

The handling of event can be implemented in many ways. One general function can collect all events and pass action as a property of event argument and based on the action through switch invoke methods for this action.

The reason why this approach was chosen is just personal. It is easier to maintain and understand the code at least for me as the developer of the code.

On the figure 55 is shown the code of **V1Listener\_Update** method. The method is subscribed for **V1ItemUpdated** event which comes from **V1Listener** component.

```

private void V1Listener_Updated(object sender, EntityId e)
{
    try
    {
        Lock(updateDriver);
        repository.IsRecent(e.Id);
        Debug.WriteLine("ControllerAction:V1_Updated_Start:" + e.Id);
        string almId = repository.FindItem(e.Entity, e.Id).Value.ID;
        V1Object v1Item;
        AlmObject almItem, updatedItem;
        switch (e.Entity.ToString())
        {
            case "Story":
                v1Item = versionOne.Interface.GetStory(e.Id);
                verifyService.CheckStory<V1Object>(v1Item);
                almItem = mapperService.ConvertToAlmStory(v1Item, almId);
                updatedItem = reqFactory.UpdateStory(almItem);
                break;
            case "Defect":
                v1Item = versionOne.Interface.GetDefect(e.Id);
                verifyService.CheckDefect<V1Object>(v1Item);
                almItem = mapperService.ConvertToAlmDefect(v1Item, almId);
                updatedItem = alm.DefectFactory.UpdateDefect(almItem);
                break;
            default:
                throw new UnsupportedEntityType(e.Entity);
        }
        repository.AddToRecent(almId);
        OnHistoryRecord(new PutToHistory(new HistoryItem(Constants.Action.Update, v1Item, true)));
        repository.UpdateItem(e.Entity, v1Item, almItem);
        Debug.WriteLine("ControllerAction:V1_Updated_End:" + e.Id);
    }
    catch (RequiredFieldMissing<V1Object> ex)
    {
        HistoryItem hItem = new HistoryItem(Constants.Action.Update, ex.Item, false);
        hItem.SetMissingField(ex.MissingField);
        OnHistoryRecord(new PutToHistory(hItem));
    }
    catch (KillEventException ex)
    {
        Debug.WriteLine("ControllerAction:V1_Updated_KillEventExceptionHandled:" + e.Id);
        return;
    }
    finally
    {
        Unlock(updateDriver);
    }
}

```

Figure 55 – Code of V1Listener\_Update method

To make sure that every event finishes without interrupts **EventWaitHandle** is implemented. At the beginning of event the EventWaitHandle method WaitOne is called to make sure that thread will be locked until the operation is not finished at the end of operation method Set is called and thread is available for events. If there is more events raised at the same time, controller will serve the first request and the other put into queue and as controller thread became available it takes item from queue with priority until the queue is empty.

The logic of locking and unlocking of thread is done through **Lock** and **Unlock** methods which can be seen on figure 56.

```
private void Lock(EventWaitHandle handle)
{
    handle.WaitOne();
    Debug.WriteLine("ControllerAction:Lock_EventWaitHandler");
}

private void Unlock(EventWaitHandle handle)
{
    handle.Set();
    Debug.WriteLine("ControllerAction:Unlock_EventWaitHandler");
}
```

Figure 56 – Methods for locking and unlocking of thread

### 5.6.5 Mapper Service

Mapper service works as a bridge between objects of V1 (V1Object) and ALM (ALMObject), mapper is able to transform V1Object to AlmObject and vice versa.

From given requirements, mapper needs to contain information about fields how they are represented at both tools, which means that mapper needs be able to distinguish between ID and name of the field on the V1 side and between label and database name on the ALM side. Also needs to contain links between those fields to be able transform one object to another.

Methods for mapping values of one object to another are called **ConvertToAlmObject** and **ConvertToV1Object** and can be seen at figure 57. Those methods are generic and used for mapping of common fields. That is the reason why the access modifier is private.

Because most of the entities have some special attributes which needs to be mapped specially for the entity there are public functions which implement this (e.g. ConvertToAlmStory).



An example of mapping method for mapping of Story entity is shown below on figure 58.

```
private AlmObject ConvertToAlmObject(V1Object item, Dictionary<Field, Field> linkage)
{
    AlmObject almItem = new AlmObject() { Name = item.Name };
    foreach (KeyValuePair<Field, Field> fields in linkage)
    {
        string value;
        if (fields.Key.IsRelation)
        {
            value = fields.Key.Relation.FirstOrDefault(x => x.Value.ToLower() == item.GetValueByKey(fields.Key.Id).ToLower()).Key;
            value = (value == null ? "" : value);
        }
        else
        {
            value = item.GetValueByKey(fields.Key.Id);
        }
        almItem.SetValue(fields.Value.Id, value);
    }
    return almItem;
}

private V1Object ConvertToV1Object(AlmObject item, Dictionary<Field, Field> linkage)
{
    V1Object v1Item = new V1Object() { Name = item.Name };
    foreach (KeyValuePair<Field, Field> fields in linkage)
    {
        string value;
        if (fields.Key.IsRelation)
        {
            value = fields.Key.Relation.FirstOrDefault(x => x.Key.ToLower() == item.GetValueByKey(fields.Value.Id).ToLower()).Value;
            value = (value == null ? "" : value);
        }
        else
        {
            value = item.GetValueByKey(fields.Value.Id);
        }
        v1Item.SetValue(fields.Key.Id, value);
    }
    return v1Item;
}
```

Figure 57 – Common mapping methods

```
public AlmObject ConvertToAlmStory(V1Object story, string almId = null)
{
    AlmObject almItem = ConvertToAlmObject(story, fieldLinkage);
    string v1Release = story.GetValueByKey("Scope.Name");
    string v1Sprint = story.GetValueByKey("Timebox");
    string sprintId = reqStructure.GetNodes().Find(x => x.V1_Id == v1Sprint && x.ParentNode.Name.EndsWith(v1Release)).Alm_Id;
    almItem.ParentID = sprintId;
    SetId(ref almItem, almId);
    return almItem;
}

private void SetId(ref AlmObject item, string almId)
{
    if (almId != null)
        item.ID = almId;
}
```

Figure 58 – Mapping method for story entity from V1 to ALM

With the transforming of fields comes few difficulties as was mentioned before V1 works with type of field which is called relation. It needs to convert the field value to value ID and vice versa depends on the direction of transforming. How this difficulty is handled can be seen at figure 57 where if condition checks if field type is relation (IsRelation).

### 5.6.6 Verify Service

Verify service make sure that entities before it is own synchronization are properly filled in and prepared for synchronization.

What **properly filled** in means? On created or updated event are entities downloaded with fields which are user-defined at configuration file. If user forgot to fill some field and this field is required, the verify service throw exception with missing field.

What **properly prepared** means? Objects are returned from mapper service as entities of other type based on linkages of fields which are defined by user at configuration file, verify service does not allow passing those uncompleted entities to factory.

Idea behind creation of verify service is substitute REST API verify engine to get more transparent view what is happening at synchronizer or why is synchronization process failing.

As REST is based on HTTP calls the user is notified about a failure through HTTP return code. But those problems with not properly filled or prepared entity returns in most of the cases just RC 500 which represents an internal error.

With response containing RC 500 VersionOne REST API and ALM REST API give you information that there was some problems with parsing entity or that you are not able to create entity, but no reason why. And there can be millions of reasons e.g. call to bad endpoint, parent ID does not exists, not filled required field. So the transparency of synchronization was the point.

At figure 59 are shown methods to check if entity is properly filled in. The first method CheckStory just wraps the second method which is more general. The wrapping is done because of usability. The method are created as generic because we can check the V1Object and ALMObject depends on the synchronization flow. Based on the type of entity is chosen the list of required fields for this entity.

```

public void CheckStory<T>(T item)
{
    CheckEntity<T>(item, Constants.Entities.Story);
}

private void CheckEntity<T>(T item, Constants.Entities type)
{
    Dictionary<string, Dictionary<Field, Field>> requiredFields;
    switch (typeof(T).Name)
    {
        case "V1Object":
            requiredFields = almRequiredMap; // item will be created at ALM so needs to be checked for ALM REQ FIELDS
            break;
        case "AlmObject":
            requiredFields = v1RequiredMap; // item will be created at V1 so needs to be checked for V1 REQ FIELDS
            break;
        default:
            throw new Exception("Unable to resolve required fields for entity");
    }
    foreach(KeyValuePair<Field,Field> reqMapping in requiredFields[type.ToString()])
    {
        Field reqField = typeof(T).Name.ToString() == "V1Object" ? reqMapping.Key : reqMapping.Value;
        object value = item.GetType().GetMethod("GetValueByKey").Invoke(item, new object[] { reqField.Id });
        if (value == null)
        {
            throw new RequiredFieldMissing<T>(item, reqMapping);
        }
        else if(((string)value).Length > 0)
        {
            continue;
        }
        else{
            throw new RequiredFieldMissing<T>(item, reqMapping);
        }
    }
}

```

Figure 59 – Method which checks if Story entity is properly filled in

### 5.6.7 Mail Service

Mail service is responsible for sending emails to subscribed users. The subscription of users is done at configuration part, for more information check subsection 5.5.6.

The mail service should be able to inform user that something goes wrong with his instance and that he needs to check synchronizer itself or just the entities.

The service is implemented to catch exceptions which can abort whole synchronization or abort a step of synchronization because synchronizer is designed as server application and we do not suppose that it will run on user local machine.

This approach can cause some troubles to user because for user the synchronizer will work like black-box and if something fails user will not recognize it until he notices that information are not shared anymore.

To avoid this problem you can define subscribers at configuration part and those people will receive emails on synchronizer errors and will be able to solve those problems.

Right now is the email service hooked up to OnHistoryRecord event, because the arguments of this event keep information if synchronization was successful.

The trigger of the mail service is shown at figure 60. Method used for sending the emails is shown at figure 61.

```
private void OnHistoryRecord(PutToHistory e)
{
    if( !e.Item.WasSucessfullySynchronized ){
        mailService.SendEmail("Synchronization failed - ItemID : " + e.Item.ID, e.Item.ToString());
    }
    EventHandler<PutToHistory> handler = HistoryRecord;
    if (handler != null)
    {
        HistoryRecord(this, e);
    }
}
```

Figure 60 – The trigger of mail service

```
public void SendEmail(string subject, string body)
{
    foreach (MailAddress subscriber in subscribers)
    {
        using (var message = new MailMessage(mailServiceAdress, subscriber)
        {
            Subject = subject,
            Body = body
        })
        {
            client.Send(message);
        }
    }
}
```

Figure 61 – Method for sending emails

## 5.6.8 Repository

The repository is place where synchronizer keeps entity information. Right now is stored just information about IDs of linked entities and those information are stored in cache memory, for refreshing those links if you turn synchronizer off is responsible Initialization service described above at subsection 5.6.1.

The repository class is prepared to be extended pretty easy because repository defines just interface not the implementation behind. Which means that for adding the database just DAO layer needs to be implemented behind this interface.

The reason why repository was implemented with just caching of ID links is because both tools stores entities inside its own database and those databases are pretty big. So there is no reason to create another database with the same data.

### 5.6.9 GenericObject

The **GenericObject** is abstract class which defines default properties and key/value container which store field values based on the ID of field as key and field value as value of that key. The class diagram is shown at figure 62.

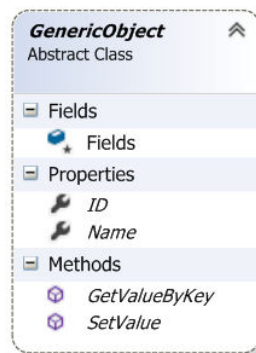


Figure 62 – GenericObject abstract class

From the GenericObject inherits objects which represent ALM and V1 entities. Class is abstract because as mentioned before IDs of fields in both tools needs to be defined.

The **AlmObject** represents ALM entities for synchronizer and is created by XML parser which sits on the top of ALM REST Client and converts XML response to AlmObject.

The **V1Object** represents V1 entities for synchronizer. The object is created by JSONParser which parses the response from calls to **query.v1** endpoint.

The values are stored at Dictionary class which is part of GenericObject. Representing objects above specify properties with the IDs of the fields. At the figure 63 are shown class diagrams of objects.

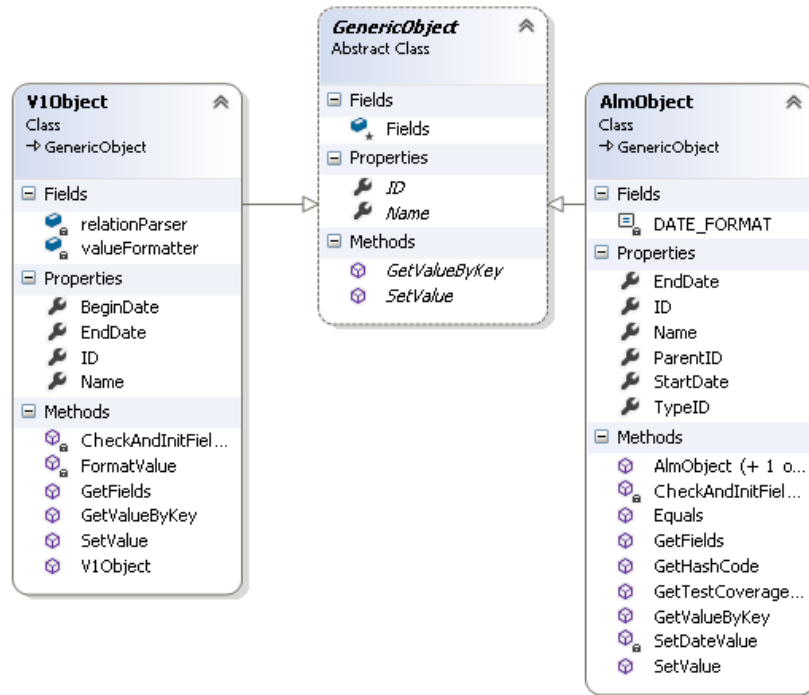


Figure 63 – Class diagram of the objects

## 5.6.10 Workflow

This section will provide us the general overview on event workflow (Figure 64) through the synchronizer core components.

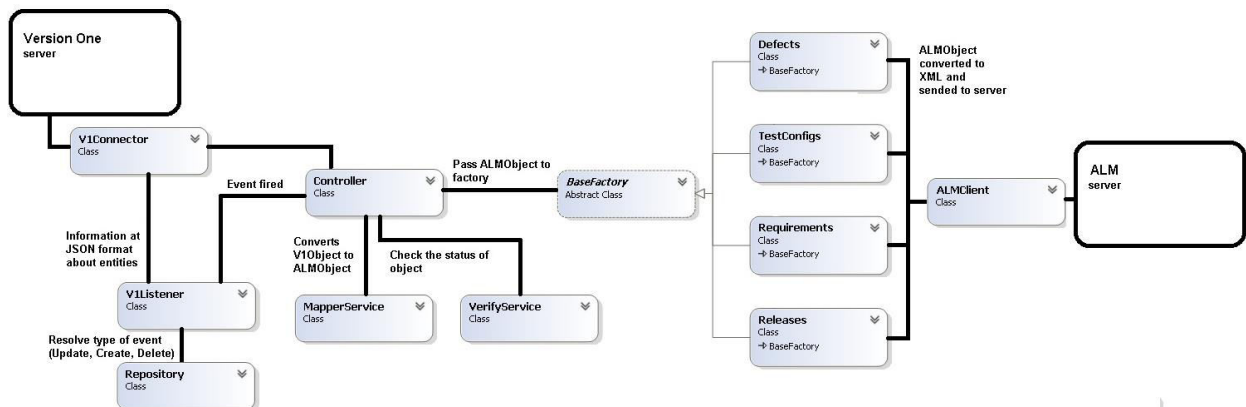


Figure 64 – Workflow through synchronizer core

At the diagram above can be seen the whole process of handling the information about the changes which were made at VersionOne tool.

Everything starts at V1Listener which gets the information at JSON format about the changes for last minute. V1Listener analyze the information and ask repository if this entity

already exist and needs to be updated or if it is the new entity or if the entity was deleted. Based on that V1Listener prepares event with all needed event argument and triggers the event.

Triggered event is captured by controller which use the V1Connector to get all fields specified by user at the configuration file. As controller gets response, converts it to V1Object. Then the object is passed to MapperService where the transformation from V1Object to ALMObject occurs. As controller receive the ALMObject from MapperService it will pass it to VerifyService which checks if the ALMObject contains all fields and values needed for successful interaction with ALM server.

If VerifyService checks the object and no problem was found then the object is passed to the right factory, based on the entity type.

## 5.7 Synchronizer instance manager

### 5.7.1 Instance Process

The **InstanceProcess** class collects data about synchronization process from all modules and presents them through instance manager to the user. User is also able to start/stop synchronization process.

The class diagram of InstanceProcess is shown on figure 65

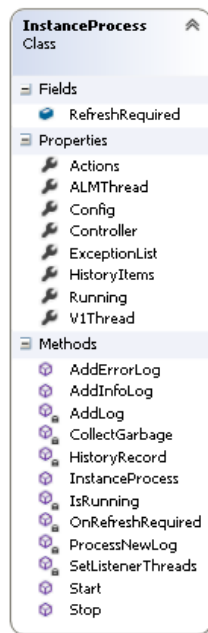


Figure 65 – Instance process class diagram

From class diagram is easy to find out what information are stored and available through instance manager.

The most important stored information is configuration file (config variable), listener threads (ALMThread and V1Thread variables), controller (Controller variable), ExceptionList and HistoryItems.

The configuration file is input parameter to constructor (Figure 66) of InstanceProcess class, so InstanceProcess is unique and defined by it is configuration. Other information are passed to controller when synchronization is already running.

```
public InstanceProcess(SyncConfiguration config)
{
    this.Config = config;
    this.Actions = new List<string>();
    this.ExceptionList = new List<string>();
    this.HistoryItems = new List<HistoryItem>();
}
```

Figure 66 – Constructor of InstanceProcess class

The function responsible for starting of synchronization process is shown at figure 67.

```
public void Start()
{
    if (!IsRunning())
    {
        try
        {
            Controller = new Controller(Config);
            Controller.ListenerInitialized += new EventHandler<Events.ListenerInitialized>(SetListenerThreads);
            Controller.LogEvent += new EventHandler<Events.Log>(ProcessNewLog);
            Controller.HistoryRecord += new EventHandler<Events.PutToHistory>(HistoryRecord);
            Controller.Start();
        }
        catch (ConnectionAborted cEx)
        {
            MessageBox.Show("User aborted connecting to " + cEx.Source.ToString() + Environment.NewLine +
                "Unable to reach " + cEx.Source.ToString() + " site", "Connection error!",
                MessageBoxButtons.OK, MessageBoxIcon.Stop);
            return;
        }
    }
}
```

Figure 67 – Function responsible for starting synchronization



At the figure 67 we can see that if synchronization process is not running new controller is created with configuration file which is set to InstanceProcess at constructor.

After creation of controller InstanceProcess class subscribes to controller events ListenerInitialized, LogEvent and HistoryRecord then synchronization starts.

The ListenerInitialized event is raised from controller as listener threads are initialized and calls SetListenerThread function (Figure 68) at InstanceProcess class.

```
private void SetListenerThreads(object sender, Events.ListenerInitialized e)
{
    if (e.Source == Constants.Source.V1)
    {
        V1Thread = e.ListenerThread;
    }
    if (e.Source == Constants.Source.ALM)
    {
        ALMThread = e.ListenerThread;
    }
}
```

Figure 68 – Function which sets listener thread to InstanceProcess class

The event argument is ListenerInitialized and class diagram of it is shown at figure 69.

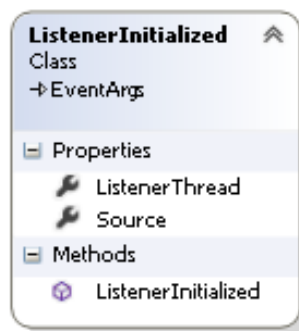


Figure 69 – ListenerInitialized event argument

It contains only information about source, which listener was initialized and the thread of the listener.

The LogEvent event is fired if any action needs to be logged and available to user. The event invokes ProcessNewLog function (Figure 70) which stores the event at the right list. Right now there is only one type of log and it is the ERROR type. But should not be problem to expand it with another type of log in future if needed.

The AddErrorLog function just calls function with right list where should be error log stored.

The AddLog function stores the log information and calls OnRefreshRequired event. This event cause refresh of instance manager UI if it is opened and active.

```
private void ProcessNewLog(object sender, Events.Log e)
{
    if (e.LogType == Constants.LogType.ERROR)
        AddErrorLog(e.Message);
}

public void AddErrorLog(string log)
{
    AddLog(log, ExceptionList);
}

private void AddLog(string log, List<string> list)
{
    list.Add(log);
    OnRefreshRequired(EventArgs.Empty);
}
```

Figure 70 – Functions to log actions

The HistoryItem event (Figure 71) is raised as synchronizer ends with synchronization process. The arguments of the event holds information about it is own synchronization if was successful or if not what can cause the problem. Also calls OnRefreshRequired event for refreshing the instance manager UI.

```
private void HistoryRecord(object sender, Events.PutToHistory e)
{
    HistoryItems.Add(e.Item);
    OnRefreshRequired(EventArgs.Empty);
}
```

Figure 71 – Function to store the synchronization attempts

## 6 Jenkins plugin – Dingo

The Jenkins is a tool for continuous integration which can interact with SCM (Source Code Management tools) download the non-compiled code, compile code, invoke tests etc.

Everything interaction is done in the form of plugins, so you are able to invoke maven, ant and much more. The important for us is that Jenkins allows us to invoke JUnit, NUnit or xUnit tests and store them. For more information about Jenkins checks chapter four.

The idea was to share the results from tests invoked by Jenkins to HP ALM. This should be done after the build is finished, if the build have assigned tests to it.

The Jenkins supports Java language for development of backend actions and use jelly for frontend.

**Dingo** is the name of the plugin.

### 6.1 Research

As I have done some research of Jenkins tool and also other tool possibilities. I noticed that there are some other tools with same functionality (TeamCity for example) and they are also used at CA Technologies.

So my idea was to create some generic part of the plugin which will work on collecting the test results from the storage specified by implementation and the part where will be the own implementation for given tool so if we would like to extend this plugin also for TeamCity we will need to code just the implementation and not the part of extracting and sending the data.

## 6.2 Architecture

From the idea above I came up with the two component architecture the first component is **Dingo\_Core** which is generic part of the plugin and the second is **Jenkins\_Dingo** component where the tool implementation is specified.

The **Dingo\_Core** component is responsible for collecting the results from Jenkins and converting them to an object which ALMClient is able to understand. As those objects are prepared the pre-defined structure is created inside ALM. This part is written in the Java language and gets distributed as jar.

The **Jenkins\_Dingo** component specifies the implementation of the plugin inside Jenkins. Plugin offers it its own functionality to user as post-build action, it means that Dingo\_Core process will be invoked after the build execution. This part is written also in Java, the second responsibility of that component is implement the frontend of the plugin.

## 6.3 Pre-defined structure

The term predefined structure was mentioned at subsection about the architecture of plugin it is basically the conversion between the Jenkins test suites and test cases to HP ALM TestPlan and TestLab structure.

On the structure we co-worked with the various teams. We collect the information about how they use the TestPlan and TestLab modules at ALM and also if they have any idea or request how the predefined structure should look like.

The figure 72 shows how the test result report (NUnit) looks like.

```

▼<testResult>
  <duration>0.143</duration>
  <empty>>false</empty>
  <failCount>1</failCount>
  <passCount>3</passCount>
  <skipCount>0</skipCount>
  ▼<suite>
    ▶<case>...</case>
    ▶<case>...</case>
    ▼<case>
      <age>1</age>
      <className>calc.Calctest</className>
      <duration>0.048</duration>
      ▼<errorStackTrace>
        MESSAGE: Expected: 2 But was: 3.0d ++++++
        STACK TRACE: at calc.Calctest.TestDivide()
        in c:\Jenkins\workspace\Calculator_dotNET\calc\Calctest.cs:line 41
      </errorStackTrace>
      <failedSince>27</failedSince>
      <name>TestDivide</name>
      <skipped>>false</skipped>
      <status>REGRESSION</status>
    </case>
    ▼<case>
      <age>0</age>
      <className>calc.Calctest</className>
      <duration>0.001</duration>
      <failedSince>0</failedSince>
      <name>TestMultiply</name>
      <skipped>>false</skipped>
      <status>PASSED</status>
    </case>
  </suite>
</testResult>

```

Figure 72 – NUnit test result report

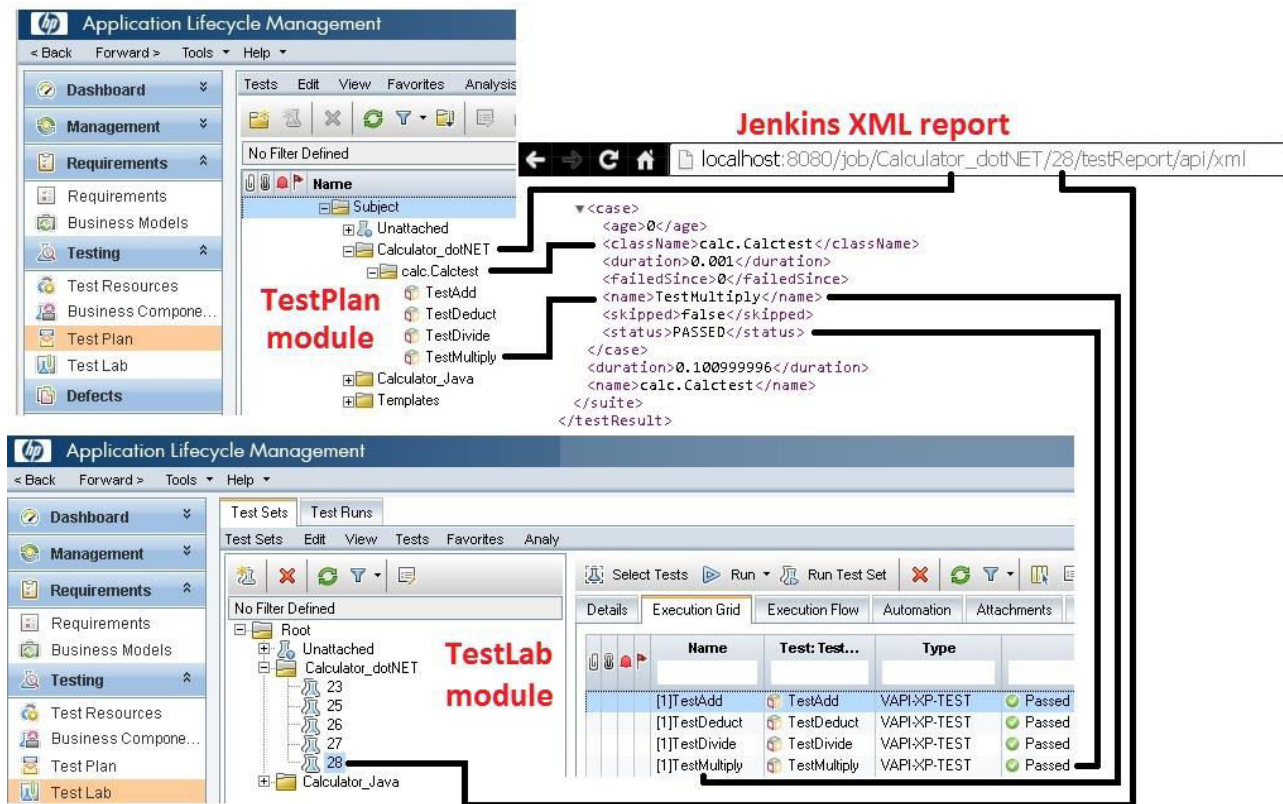
The information which we are interested in are shown at table 3.

Path (nodes taken from testResult)	Information
duration	The duration of all tests at build
failCount	Number of failed tests at build
passCount	Number of passed tests at build
skipCount	Number of skipped tests at build
suite	Test suite of the whole build
Suite/duration	Duration of the test suite
Suite/name	Name of the test suite
suite/case	Test case of the give test suite

suite/case/className	Name of the test suite
suite/case/duration	Duration of the test case
suite/case/errorStackTrace	Error message for failed test case
Suite/case/failedSince	Number of the build since test failing
Suite/case/name	Name of the test case
Suite/case/status	Status of the test case

**Table 3 – Data from XML NUnit report**

On the figure 73 will be shown creation of predefined structure. It means how are the data from XML are converted to TestLab and TestPlan module at ALM.



**Figure 73 – Conversion of XML to ALM structure**

## 6.4 Dingo Core

The Dingo\_Core functionality was described before, so I will just summarize it there for better imagination. The component is responsible for picking up the test result reports from any continuous integration tool which is able to present the results at XML format.

Those result reports are converted to objects which are able to be represented by ALM and as those presentable objects are prepared the pre-defined structure is created at ALM.

On the figure 73 can be seen the structure of the project.

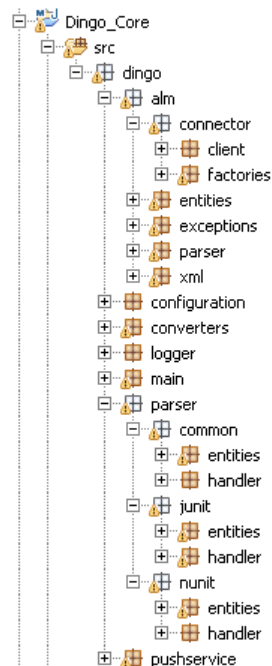


Figure 73 – The structure of Dingo\_Core

Now the each package will be described with some overview about functionality and also some code snippets. Right now the plugin support results only from JUnit, NUnit or xUnit frameworks.

For xUnit are used the JUnit classes because the report is the same as for JUnit.

### 6.4.1 ALM Client

The client package (Figure 74) contains classes responsible for interacting with ALM server through REST calls.

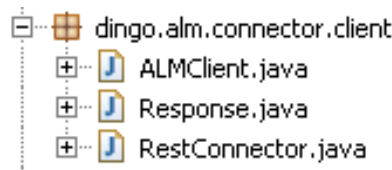


Figure 74 – Classes of ALM client

The **ALMClient** allows to login, logout, createQCSession token (ALM 12 support), check if user is authenticated (Figure 75) and also the factories are accessible through it.

The client is wrapping RestConnector class because for all operations mentioned above are REST calls needed.

```
public String isAuthenticated() throws Exception {
    String isAuthenticatedUrl = con.buildUrl("rest/is-authenticated");
    String ret;
    Response response = con.httpGet(isAuthenticatedUrl, null, null);
    int responseCode = response.getStatusCode();
    if (responseCode == HttpURLConnection.HTTP_OK) {
        ret = null;
    }
    else if (responseCode == HttpURLConnection.HTTP_UNAUTHORIZED) {
        Iterable<String> authenticationHeader =
            response.getResponseHeaders().get("WWW-Authenticate");

        String newUrl =
            authenticationHeader.iterator().next().split("=")[1];
        newUrl = newUrl.replace("\\\"", "");
        newUrl += "/authenticate";
        ret = newUrl;
    }
    else {
        throw response.getFailure();
    }
    return ret;
}
```

Figure 75 – Function to check if user is authenticated

The function use HTTP GET request to the rest/is-authenticated endpoint. If the user is not authenticated function returns the URL of endpoint as a string where the user should authenticate. If the user is authenticated function returns null.



The **Response** represents the XML response from server as object, so it is much easier to get the data from response.

The **RestConnector** is able to create supported HTTP calls (Get, Put, Post and Delete) with all mandatory parameters.

## 6.4.2 ALM Factories

The factories package (Figure 76) contains the classes responsible for creating objects inside the ALM server through REST calls available from RestConnector class mentioned above.

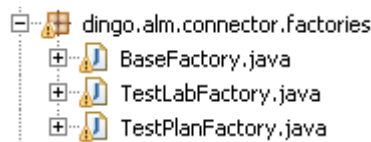


Figure 76 – Classes of factories package

The **BaseFactory** is abstract class for all factories as the plugin works only at TestLab and TestPlan module the factories for those modules are implemented only.

The abstract class wraps the RestConnector class and adds the headers which are related to operations with entities. Also contains the parser for parsing the XML responses to objects and entity builder which is responsible for creating the XML file from object in the format that reflects the ALM XML standards.

The usage of base factory will be described on function getEntities (Figure 77).

```
protected List<ALMObject> getEntities(String entity) throws Exception{
    String collectionUrl = con.buildEntityCollectionUrl(entity);
    Map<String, String> requestHeaders = new HashMap<String, String>();
    requestHeaders.put("Accept", "application/xml");
    String xmlResponse = con.httpGet(collectionUrl, "", requestHeaders).toString();
    return parser.parse(xmlResponse);
}
```

Figure 77 – Function to get collection of entities

The function input parameter is entity which specifies the name of the collection we want to reach. The URL of collection is returned from buildEntityCollectionUrl (Figure 78), headers are created after the URL and are represented as HashMap by key (header name) and value (header value).

```

public String buildEntityCollectionUrl(String entityType) {
    return buildUrl("rest/domains/"
        + domain
        + "/projects/"
        + project
        + "/"
        + entityType
        + "s");
}

```

Figure 78– Function to building the URL of REST endpoint

In our case the header specifies the format of the response which is set as XML format. Then HTTP GET request is invoked to the URL of collection with no request body (GET request) but with defined headers (e.g. Accept). Response is then parsed into collection of ALMObjects.

The **TestLabFactory** is responsible for CRUD operations at TestLab module at ALM. The available entities inside the TestLab are shown in table 4.

Name	REST endpoint (rest/domains/\$DOMAIN/projects/\$PROJECT/...)
Test Set	../test-sets
Test Set folder	../test-set-folders
Test Instance	../test-instances
Run	../runs
Run Step	../runs/\$RUN_ID/run-steps

Table 4 – Entities and endpoints available at TestLabFactory

At the BaseFactory the getEntities function was shown, but as BaseFactory is just abstract class and cannot be instantiated the getTestSetFolders method (Figure 79) will demonstrate the usage.

```

private final String TEST_SET_FOLDER = "test-set-folder";

public List<ALMObject> getTestSetFolders() throws Exception{
    return getEntities(TEST_SET_FOLDER);
}

```

Figure 79 – Method to get all Test Set folder entities

The part of the factory also defines the endpoint of the entity. If you noticed small difference that the “s” is missing (test-set-folder/test-set-folders) at definition you are right. But if you check the functionality of buildEntityCollectionUrl function (Figure 78) you will see that function automatically adds “s” at the end of the URL.

The feature is there to keep consistent name convention because the entity name is Test Set Folder and not Test Set Folders. And the function just invokes the getEntities with the right REST endpoint name.

The **TestPlanFactory** is responsible for CRUD operations for TestPlan module at ALM. The available entities to that module are shown at table 5.

Name	REST endpoint (rest/domains/\$DOMAIN/projects/\$PROJECT/...)
Test Folder	../test-folder
Test	../test
Test Configuration	../test-config

Table 5 – Entities and endpoints available at TestPlan factory

As for factory above the usage of inherited function getEntities from BaseFactory is shown on figure 80.

```
private final String TEST_FOLDER = "test-folder";

public List<ALMObject> getTestFolders() throws Exception{
    return getEntities(TEST_FOLDER);
}
```

Figure 80 – Method to get all Test Folder entities

### 6.4.3 ALM Entities

The Entities package (Figure 81) contains only one class ALMObject.



Figure 81 – Classes of entities package

The **ALMObject** is class that represents ALM entities for plugin. As ALMObject is written in really generic way it is able to represent every entity of ALM. The values of fields are stored at HashMap (Figure 82) which is defined with the key (field name) and value (value of the field).

```
private HashMap<String, String> values;
```

Figure 82 – HashMap for ALM values

And the most important methods of the ALMObject class are setValues, getValue, updateValue and getXml. All methods mentioned above can be seen on figure 83.

```
public void setValues(HashMap<String,String> map){
    this.values = map;
}

public String getValue(String key){
    if ( values.containsKey(key) )
        return values.get(key);
    else
        return null;
}

public void updateValue(String key,String value){
    if ( values.containsKey(key) ){
        values.remove(key);
    }
    values.put(key, value);
}

public String getXml(String entityName){
    StringBuilder sb = new StringBuilder();
    sb.append("<Entity Type=\""+ entityName + "\"><Fields>");
    for(Map.Entry<String, String> attribute : values.entrySet()){
        sb.append("<Field Name=\""+ attribute.getKey()+"\">"
            + "<Value>"
            + attribute.getValue()
            + "</Value>"
            + "</Field>");
    }
    sb.append("</Fields></Entity>");
    return sb.toString();
}
```

Figure 83 – Most important methods of ALMObject class

The set, get and update functions are pretty self-explanatory. The getXml function generates the string in the XML format by ALM standards.

For creation of ALM standardized XML format is used StringBuilder class, let's take a look at the code. The first append is for defining the entity type and opening the fields node then there is foreach cycle through all HashMap values and at the end fields and entity node is closed.

The function functionality is visualized at figure 84.

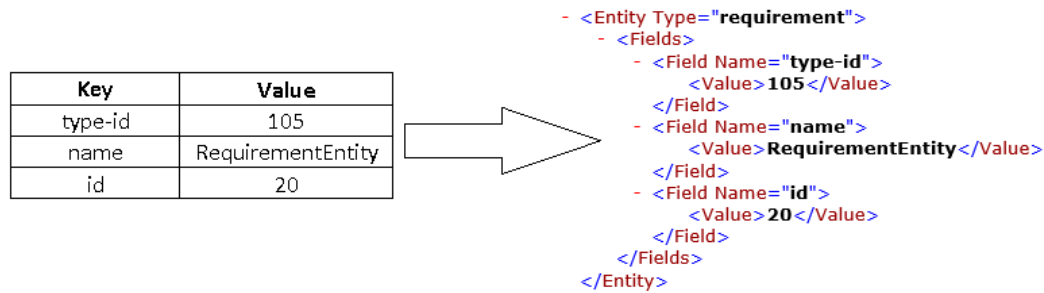


Figure 84 – Visualization of getXml function

#### 6.4.4 ALM Parser

The Parser package (Figure 85) contains the classes responsible for parsing the ALM responses at XML format to ALMObject.

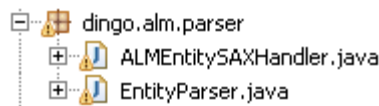


Figure 85 – Classes of parser package

The **ALMEntitySAXHandler** is class that inherits from existing java DefaultHandler class, which defines the events that can be handled when reading the XML file.

ALMEntitySAXHandler class then defines the actions on those events. In our case we mostly care just about the start and end of element event. The actions are shown on figure 86.

```

@Override
public void startElement(String uri, String localName, String qName,
    Attributes attributes) throws SAXException {
    this.node = qName;

    switch(qName){
        case "Entity":
            object = new ALMObject();
            break;
        case "Fields":
            values = new HashMap<>();
            break;
        case "Field":
            attName = "";
            attValue = "";
            attName = attributes.getValue("Name");
            break;
        case "Value":
            break;
        default:
            break;
    }
}

@Override
public void endElement(String uri, String localName, String qName)
    throws SAXException {
    switch(qName){
        case "Entity":
            objects.add(object);
            break;
        case "Fields":
            object.setValues(values);
            break;
        case "Field":
            if ( !values.containsKey(attName) && !attValue.equals(""))
                values.put(attName, attValue);
            break;
        case "Value":
            break;
        default:
            break;
    }
}

```

Figure 86 – Handling the start (left) and end (right) element events

The **EntityParser** class just implements the **ALMEntitySAXHandler**. In the example on figure 87 is shown code snippet of method responsible for parsing of XML response.

```
private final SAXParserFactory factory = SAXParserFactory.newInstance();
private SAXParser saxParser;
private final ALMEntitySAXHandler handler = new ALMEntitySAXHandler();
private InputStream xmlInput;

public List<ALMObject> parse(String resp) throws SAXException, IOException{
    xmlInput = new ByteArrayInputStream(resp.getBytes(StandardCharsets.UTF_8));
    saxParser.parse(xmlInput, handler);
    return handler.objects;
}
```

Figure 87 – Code snippet of method for parsing the XML response

## 6.4.5 Configuration

The Configuration package (Figure 88) consists of classes where the user and tool information are stored.



Figure 88 – Content of configuration package

The **Config** class is placeholder for **ConfigData**, there are security reasons why the sensitive data are stored in special class, and also guide for the Jenkins developers marks this approach as best practice.

The **ConfigData** class stores information defined by user and extracted from tool. In the table 6 are shown values stored in **ConfigData** for the Jenkins implementation.

Values defined by user	Values from Jenkins
ALM URL	Jenkins URL
ALM Username	Project Name
ALM Password	Build Number
ALM Domain	Path to result report
ALM Project	

Table 6 – Overview of data stored at **ConfigData** class

### 6.4.6 Logger

The Logger package (Figure 89) contains classes responsible for logging actions through the process of collecting the result reports and pushing them to ALM.



Figure 89– Classes of Logger package

The **Logger** class is just general class which is prepared to be specified by the tool logging service. As the most of web application log service is built around PrintStream.

Class is using method setLogger (Figure 90) for communication with tool logging service.

```
public static void setLogger(PrintStream stream){
    logger = stream;
}
```

Figure 90 – Method sets reference of web log service to plugin log service

New logs are added through addLog method (Figure 91). The log is added to collection and if any PrintStream is referenced the value is also printed to the web log service.

```
public static void addLog(String log){
    logs.add(log);
    if ( logger != null)
        logger.println(log);
}
```

Figure 91 – Add new log to collection and stream

### 6.4.7 Common entities

The Entities package (Figure 92) contains the abstract (general) definition of available nodes from test result report hierarchy (Figure 93).

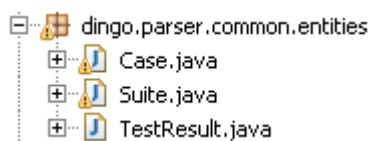


Figure 92 – Classes of entities package

```

▼<testResult>
  <duration>0.143</duration>
  <empty>false</empty>
  <failCount>1</failCount>
  <passCount>3</passCount>
  <skipCount>0</skipCount>
  ▼<suite>
    ▶<case>...</case>
    ▶<case>...</case>
    ▶<case>...</case>
    ▶<case>...</case>
    <duration>0.143</duration>
    <name>calc.Calctest</name>
  </suite>
</testResult>

```

**Figure 93 – Test result report hierarchy**

The **TestResult** class represents whole report and it can contains many of test suites under.

Available attributes of TestResult class are shown at table 7.

Attributes
duration
empty
failCount
passCount
skipCount
suites

**Table 7 – Attributes of TestResult class**

The **Suite** class represents test suites and it can contains many of test cases under. Available attributes of Suite class are shown at table 8.

Attributes
duration
name
cases

**Table 8 – Attributes of Suite class**



The **Case** class represents each test case and it is the node of test result report which the plugin is most interested in.

Available attributes of Case class are shown at table 9.

Attributes
age
className
duration
failedSince
name
skipped
status

Table 9 – Attributes to Case class

#### 6.4.8 Common handler

The **Handler** package (Figure 94) contains abstract class which defines the handler for parsing the XML.



Figure 94 – Handler package

The **SAXHandler** is abstract handler which extends the existing java DefaultHandler. For the parsing of XML file is used SAXParser (Simple API for XML).

In the SAXHandler (Figure 95) is just predefined what events are we interested in from DefaultHandler and needs to be implemented for parsing the test result report.

```
public abstract class SAXHandler extends DefaultHandler {

    protected String node;
    public abstract void startDocument() throws SAXException;
    public abstract void endDocument() throws SAXException;
    public abstract void startElement(String uri, String localName, String qName, Attributes attributes) throws SAXException;
    public abstract void endElement(String uri, String localName, String qName) throws SAXException;
    public abstract void characters(char ch[], int start, int length) throws SAXException;
    public abstract void ignorableWhitespace(char ch[], int start, int length) throws SAXException;

}
```

Figure 95 – Events for parsing the test result report

### 6.4.9 JUnit entities

The JUnit entities package (Figure 96) extends the classes from dingo.parser.common.entities for the JUnit needs.

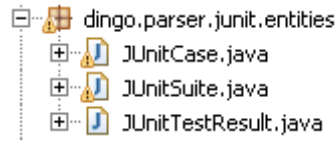


Figure 96 – Package with JUnit entities

The **JUnitTestResult** class extends the common TestResult class with the JUnit specific attributes. As the TestResult of JUnit does not offer any specific attributes, the class is there to keep consistent naming convention through the plugin.

The **JUnitSuite** and **JUnitCase** class extends the common Suite and Case class with JUnit specific attributes.

At the table 10 are shown specific JUnit attributes for suite.

Attributes
stderr
stdout

Table 10 – JUnitSuite and JUnitCase specific attributes

### 6.4.10 JUnit handler

The JUnit handler package (Figure 97) contains classes responsible for handling the process of parsing JUnit test result report.

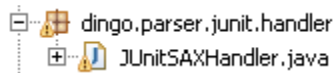


Figure 97 – Content of JUnit handler package

The **JUnitSAXHandler** class extends the SAXHandler class from dingo.parser.common.handler which is abstract class with defined events without any action assigned to defined events.

The most interesting events for the plugin are **startElement** (Figure 98) which is invoked if a node is opened, then **characters** (Figure 99) for values stored at nodes and **endElement** (Figure 100) if a node is closed.

```
@Override
public void startElement(String uri, String localName, String qName,
    Attributes attributes) throws SAXException {
    node = qName;
    switch(qName){
        case "testResult":
            result = new JUnitTestResult();
            values = new HashMap<>();
            break;
        case "suite":
            if ( !result.isSet() )
                result.setValues(values);
            testSuite = new JUnitSuite();
            values = new HashMap<>();
            break;
        case "case":
            testCase = new JUnitCase();
            values = new HashMap<>();
            break;
        default:
            break;
    }
}
```

Figure 98 – startElement event with defined action

```
@Override
public void characters(char[] ch, int start, int length)
    throws SAXException {
    String value = new String(ch, start, length).trim();
    if(value.length() == 0)
        return;
    if ( values.containsKey(node) ){
        String existingValue = values.get(node);
        values.remove(node);
        value = existingValue + value;
    }
    values.put(node,value);
}
```

Figure 99 – characters event with defined action

```

@Override
public void endElement(String uri, String localName, String qName)
    throws SAXException {

    switch(qName){
    case "testResult":
        break;
    case "suite":
        testSuite.setValues(values);
        result.addSuite(testSuite);
        values = new HashMap<>();
        break;
    case "case":
        testCase.setValues(values);
        testSuite.addCase(testCase);
        values = new HashMap<>();
        break;
    default:
        break;
    }
}

```

Figure 100 – endElement event with defined action

### 6.4.11 NUnit entities

The NUnit entities package (Figure 101) extends the classes from `dingo.parser.common.entities` to NUnit needs.

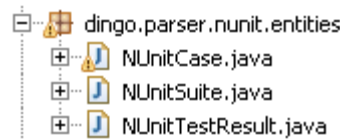


Figure 101 – Content of NUnit entities package

All classes are almost the same as in the case of JUnit implementation. For more information take a look for `dingo.parser.junit.entities` chapter.

### 6.4.12 NUnit handler

The NUnit handler package (Figure 102) contains classes responsible for handling the process of parsing NUnit test result report.

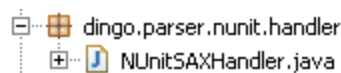


Figure 102 – Content of NUnit handler package

Implementation is also pretty same as for JUnit handler. For more information take a look for dingo.parser.junit.handler chapter.

### 6.4.13 Push service

The Push service package (Figure 103) contains classes responsible for converting the test result report to predefined structure and pushing it into ALM.

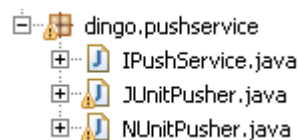


Figure 103 – Content of Pushservice package

The **IPushService** is the interface for upcoming pushers for other types of test results then JUnit, xUnit and NUnit.

The **JUnitPusher** and **NUnitPusher** class loads the results to ALM. As both pushers are similar just NUnitPusher will be described deeply. With knowledge of NUnitPusher you will be able to understand also to JUnitPusher.

The **NUnitPusher** class is responsible for loading the results from NUnit rest result report to ALM. Whole process of NUnitPusher can be divided into smaller parts – Parsing, Connecting and Pushing (loading).

The first phase is parsing process (Figure 104). There are created all mandatory items for parsing - SAXParser (Simple API for XML Parser) and NUnitSAXHandler, InputStream with test results is opened and content is passed to saxParser method parse together with handler.

```
logger.println("Parsing of test results - STARTED");
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
NUnitSAXHandler handler = new NUnitSAXHandler();
InputStream xmlInput = new URL(configuration.getData().getJenkins_result_path()).openStream();
saxParser.parse(xmlInput, handler);
logger.println("Parsing of test results - DONE");
```

Figure 104 – Process of parsing the result

If the parsing process ends without error the logger is noticed and connection phase starts. The process of connecting to ALM is shown at figure 105.

```
logger.println("Connecting to ALM...");
client = new ALMClient(configuration.getData().getAlm_url(),
    configuration.getData().getAlm_username(),
    configuration.getData().getAlm_password(),
    configuration.getData().getAlm_domain(),
    configuration.getData().getAlm_project());
boolean isConnected = client.login();
if ( !isConnected )
{
    logger.println("Unable to authenticate user!");
    return;
}
logger.println("User successfully authenticated!");
```

Figure 105 – Process of connecting to ALM

At the end of connecting phase the logger gets noticed about the result of this phase. On successfully connection the phase of pushing results to ALM starts.

The pushing phase code is shown on figure 106 and description is at table 11.

The functionality of functions with prefix “safe” is following - If item exists is returned otherwise the item is created and returned.

```
1 String projectName = configuration.getData().getJenkins_project_name();
2 String buildNumber = configuration.getData().getJenkins_build_number();
3 ALMObject testPlan_root = client.testPlanFactory.safeCreateTestFolder(projectName);
4 logger.println("TestPlan root - " + projectName + " passed!");
5 ALMObject testLab_root = client.testLabFactory.safeCreateTestSetFolder(projectName);
6 logger.println("TestLab root - " + projectName + " passed!");
7 ALMObject testLab_testset = client.testLabFactory.safeCreateTestSet(buildNumber, testLab_root.getId());
8 logger.println("TestLab TestSet - " + buildNumber + " passed!");
9 for(NUnitSuite suite : handler.result.getSuites()){
10     ALMObject suite_root = client.testPlanFactory.safeCreateTestFolder(suite.getName(),testPlan_root.getId());
11     logger.println("TestPlan TestFolder - " + suite.getName() + " passed!");
12     for(NUnitTestCase testCase : suite.getTestCases()){
13         ALMObject test_id = client.testPlanFactory.safeCreateTest(testCase, suite_root.getId());
14         logger.println("TestPlan Test - " + testCase + " passed!");
15         HashMap<String,String> filter = new HashMap<String, String>();
16         filter.put("name",test_id.getName());
17         filter.put("parent-id",test_id.getId());
18         List<ALMObject> configs = client.testPlanFactory.filterTestConfig(filter);
19         ALMObject test_instance_id = client.testLabFactory.safeCreateTestInstance(test_id,configs.get(0), testLab_testset);
20         client.testLabFactory.updateTestInstance(testCase, test_instance_id);
21         client.testLabFactory.createTestInstanceRun(test_instance_id);
22         client.testLabFactory.updateTestInstance(testCase, test_instance_id);
23         logger.println("TestPlan TestInstance - " + testCase + " passed!");
24     }
25 }
26 }
27 logger.println("Results successfully pushed to ALM!");
```

Figure 106 – Process of pushing results into ALM

Row number	Description
3	Safe creation of TestFolder at TestPlan with “project name”.
5	Safe creation of TestSetFolder at TestLab with “project name”.
7	Safe creation of TestSet with “build number” under TestSetFolder created on row 5.
9	Cycle through all test suites
10	Safe creation of TestFolder with the “name of test suite” under TestFolder created on row 3.
12	Cycle through all test cases under given suite from row 9.
13	Safe creation of Test with the “name of test case” under the TestFolder created at row 10.
15-17	Defining the conditions for filtering of existing test configs. Test configuration is automatically created as test is created. So values from the test created on row 13 are used.
18	Get results of the test filtering.
19	Safe create TestInstance of TestConfig returned on row 18
20-22	Update values of test instance from selected test case.

**Table 11 – Explanation of code at figure 95**

## 6.5 Jenkins Dingo

The Jenkins\_Dingo component connects the Jenkins with Dingo\_Core. The UI is defined here and also hooking up on events from Jenkins.

The plugin UI is stored at the file config.jelly which can be found at resources under Jenkins\_Dingo.Jenkins\_Dingo.DingoPluginController and the hooking up part can be found at Jenkins\_Dingo.Jenkins\_Dingo package and the class is name DingoPluginController.

### 6.5.1 Jelly config

The Jenkins supports Jelly framework for building UI. Jelly is framework which turns XML file into executable code.

As we need from user just minimum information the configuration UI is very simple. The content of the config.jelly file is shown on figure 107.

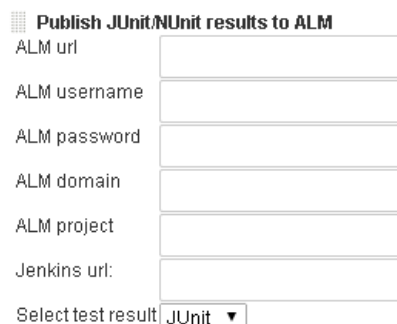
```
<?jelly escape-by-default='true'?>
<j:jelly xmlns:j="jelly:core" xmlns:st="jelly:stapler" xmlns:d="jelly:define"
xmlns:l="/lib/layout" xmlns:t="/lib/hudson" xmlns:f="/lib/form">

  <f:entry title="ALM url" field="url">
    <f:textbox />
  </f:entry>
  <f:entry title="ALM username" field="username">
    <f:textbox />
  </f:entry>
  <f:entry title="ALM password" field="password">
    <f:password />
  </f:entry>
  <f:entry title="ALM domain" field="domain">
    <f:textbox />
  </f:entry>
  <f:entry title="ALM project" field="project">
    <f:textbox />
  </f:entry>
  <f:entry title="Jenkins url:" field="jenkinsUrl">
    <f:textbox />
  </f:entry>
  <f:entry title="Select test result" field="testType">
    <select name="testType">
      <option value="junit">JUnit</option>
      <option value="nunit">NUnit</option>
    </select>
  </f:entry>
</j:jelly>
```

Figure 107 – Content of config.jelly

On the beginning the tag libraries are defined. We mostly use elements from **/lib/form** which allows us to use form element. Elements used for creation of UI are – textbox, password and select.

How Jenkins interprets the code is shown on figure 108.



**Publish JUnit/NUnit results to ALM**

ALM url

ALM username

ALM password

ALM domain

ALM project

Jenkins url:

Select test result: JUnit ▼

Figure 108 – Interpretation of config.jelly by Jenkins



## 6.5.2 Dingo plugin controller

The class controls the plugin actions in Jenkins. Request was to create plugin as post build action which means that will be invoked after the build process.

For that purpose Jenkins offers the abstract class **Recorder**, description of recorder class is shown at figure 109.

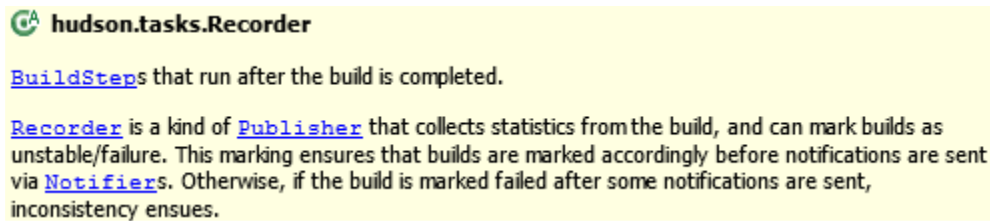


Figure 109 – Description of Recorder class

Recorder class by description perfectly suits to our plugin purpose. So `DingoPluginController` extends `Recorder` class.

To get values defined by user at configuration part we can use annotation **@DataBoundConstructor** as you may notice the package for `config.jelly` is same as for `DingoPluginController` class. Also the names of variables are same as the field attribute of entries at `config.jelly`. Because of those two restrictions we can easily pass the values to controller through constructor marked as `DataBoundConstructor` at figure 110.

```
@DataBoundConstructor
public DingoPluginController( String url,String username,
                             String password, String domain,
                             String project, String jenkinsUrl,
                             String testType) {

    this.url = url;
    this.username = username;
    this.password = password;
    this.domain = domain;
    this.project = project;
    this.jenkinsUrl = jenkinsUrl;
    this.testType = testType;
}
```

Figure 110 – Example of `DataBoundConstructor`

To invoke action at the end of the build, abstract method **perform** needs to be overridden (Figure 111).

This is the place where the Jenkins\_Dingo gets linked with the Dingo\_Core.

```
@Override
public boolean perform(AbstractBuild<?, ?> build, Launcher launcher, BuildListener listener) {
    listener.getLogger().println("Starting post build action \"" + getDescriptor().getDisplayName() + "\"");
    Config configuration = new Config(getUrl(),
                                    getUsername(),
                                    getPassword(),
                                    getDomain(),
                                    getProject(),
                                    getJenkinsUrl(),
                                    build.getProject().getName(),
                                    build.getDisplayName().replace("#", ""));

    try{
        if ( getTestType() == "junit"){
            pushService = new JUnitPusher(configuration,listener.getLogger());
        }
        else{
            pushService = new NUnitPusher(configuration,listener.getLogger());
        }
        pushService.Push();
        return true;
    }
    catch(Exception ex){
        listener.getLogger().println(ex.toString());
        return false;
    }
}
```

Figure 111 – Overridden perform method

The perform action notices the logger that action is started and configuration is created with information needed for pusher class. Then based on the type of the test the pusher class is instantiate and at the end the **Push** method (described at Dingo\_Core section) which creates predefined structure at the HP ALM is invoked.

## Conclusion

The work on the thesis gave me a lot of experience in the whole software development life cycle and because the tools was developed for company CA Technologies it gave me also knowledge about how to develop software under the company rules. I had some knowledge and experience with developing of tools before but tools were not developed for any company so I did not need to follow any rules, so this way was new for me.

The theoretical part gave me overview about existing SDLC methodologies and better understanding of used methodology at CA Technologies also I started to work for CA Technologies, for part-time right now but with possibility to switch to full-time as I graduate the reason was that they were satisfied with my knowledge and passion about software development and new technologies.

One of the most interesting parts of the thesis was creation of requirements for software because I need to get feedback from managers and then create requirements based on the feedback. Based on those feedbacks I need to come up with model and present it to them and as everyone was satisfied I started to develop it physically. This part was really new to me and I would like to say big thanks to my consultant dipl. Ing. Srdjan Nalis because lot of really useful advices about how to talk/present to managers.

The coding part was really interesting to me because I never developed software of that magnitude. I spent more than one year with the whole development process with all the negotiations etc. at the end software contains more than 10,000 lines of code. What was also new to me at coding phase was the refactoring of code. I found out that code refactoring is really needed for project of this magnitude. Also at the beginning I was not able to write clean code and I need to refactor it pretty often but with upcoming experiences I was able to write clean code much faster and I really enjoyed to make the code as much simple and readable as can be. But not always was coding just fun, there was a days that I really prey to have any other theme. On the way of coding the tools I encountered lot of problem on which I spent much time to resolve it and not always I was able to resolve it by myself. Like the problems how capture event etc. but it is described in the thesis.

At the end I would like to say thanks to CA Technologies for this opportunity.

## References

DUVALL, Paul M, Steve MATYAS a Andrew GLOVER. *Continuous integration: improving software quality and reducing risk*. Upper Saddle River, NJ: Addison-Wesley, c2007, xxxiii, 283 p. ISBN 0321336380.

User Guide. HP ALM Documentation. [online]. 1.11.2010 [cit. 2015-03-29]. Available from: [http://alm-dev.ca.com/qcbin/Help/doc\\_library/pdfs/UserGuide.pdf](http://alm-dev.ca.com/qcbin/Help/doc_library/pdfs/UserGuide.pdf)

Administration Guide. HP ALM Documentation. [online]. 1.11.2010 [cit. 2015-03-29]. Available from: [http://alm-dev.ca.com/qcbin/Help/doc\\_library/pdfs/AdminGuide.pdf](http://alm-dev.ca.com/qcbin/Help/doc_library/pdfs/AdminGuide.pdf)

REST API Reference. HP ALM Documentation. [online]. 12.4.2013 [cit. 2015-03-29]. Available from: [http://alm-dev.ca.com/qcbin/Help/doc\\_library/api\\_refs/REST/webframe.html](http://alm-dev.ca.com/qcbin/Help/doc_library/api_refs/REST/webframe.html)

OTA API Reference. HP ALM Documentation. [online]. 1.11.2010 [cit. 2015-03-29]. Available from: [http://alm-dev.ca.com/qcbin/Help/doc\\_library/api\\_refs/OTA\\_API\\_Reference.chm](http://alm-dev.ca.com/qcbin/Help/doc_library/api_refs/OTA_API_Reference.chm)

Database Model. HP ALM Documentation. [online]. 1.11.2010 [cit. 2015-03-29]. Available from: [http://alm-dev.ca.com/qcbin/Help/doc\\_library/api\\_refs/alm\\_project\\_db.chm](http://alm-dev.ca.com/qcbin/Help/doc_library/api_refs/alm_project_db.chm)

Use Jenkins. Jenkins. [online]. 6.4.2007 [cit. 2014-11-10]. Available from: <https://wiki.jenkins-ci.org/display/JENKINS/Use+Jenkins>

Plugins. Jenkins. [online]. 2.4.2007 [cit. 2014-11-10]. Available from: <https://wiki.jenkins-ci.org/display/JENKINS/Plugins>

User Guide for Enterprise and Ultimate. VersionOne. [online]. 22.4.2014 [cit. 2015-02-15]. Available from: [http://community.versionone.com/Help-Center/Getting-Started-Guides/User\\_Guide\\_for\\_Enterprise\\_and\\_Ultimate](http://community.versionone.com/Help-Center/Getting-Started-Guides/User_Guide_for_Enterprise_and_Ultimate)

Developer Library. VersionOne. [online]. 22.4.2014 [cit. 2014-09-04]. Available from: <http://community.versionone.com/Developers/Developer-Library>

BECK, Kent. *Test-driven development: by example*. Boston: Addison-Wesley, c2003, xix, 220 p. ISBN 0321146530.

MARTIN, Robert C, Michael C FEATHERS, Timothy R OTTINGER, Jeffrey J LANGR, Brett L SCHUCHERT, James W GRENNING a Kevin Dean WAMPLER. *Clean code: a handbook of agile software craftsmanship*. xxix, 431 stran. Robert C. Martin series. ISBN 978-0-13-235088-4.

EELES, Peter a Peter CRIPPS. *Architektura softwaru*. Vyd. 1. Brno: Computer Press, 2011, 328 s. ISBN 978-80-251-3036-0.

ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. *Řízení kvality softwaru: průvodce testováním*. 1. vyd. Brno: Computer Press, 2013, 208 s. ISBN 978-80-251-3816-8.