



TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

HENRIK NYMAN
MODULARIZING ROBOT DRIVER FRAMEWORK WITH A
PLUGIN ARCHITECTURE

Master of Science thesis

Examiner: Asst. Prof. Outi Sievi-Korte
Examiner and topic approved by the
Faculty Council of the Faculty of
Pervasive Computing on 15th Aug 2018

ABSTRACT

HENRIK NYMAN: Modularizing Robot Driver Framework with a Plugin Architecture
Tampere University of Technology
Master of Science thesis, 42 pages
September 12, 2018
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Asst. Prof. Outi Sievi-Korte

Keywords: Modularity analysis, Robotics, Plugin architecture, Python

Factory lines are nowadays filled with intelligent systems that can perform various tasks without human interaction. To reach the current level of intelligence and automation, The significance of software in industrial robotics has increased dramatically. That creates new challenges in software design for such environments, which are then tackled with new software platforms and frameworks.

This thesis takes an existing test automation platform and designs a new architecture based on plugins for it. The platform is used in functional testing of smartphones. The redesign aims to increase the modularity of the architecture, and thus allow for a more flexible deployment of the system in various hardware configurations.

To verify the successfulness of the new architecture, a modularity analysis is performed for both the old and the new architectures. The analysis focuses on cohesion and coupling of the classes and modules in the systems. Both the old and the new platforms are implemented in Python, so the research process will evaluate the feasibility of manual modularity analysis for a dynamically typed programming language, as these kind of analyses are usually performed on a statically typed languages utilizing static analysis tools.

The new architecture was shown to increase the cohesion, and decrease the coupling of the platform, which indicates an increase in the overall modularity of the platform. The analysis itself was found to be tedious, and the dynamic nature of Python increases the chance of errors in determining the coupling and cohesion of a component. A possibility of modifying a refactoring tool to aid in a such analysis was discussed.

TIIVISTELMÄ

HENRIK NYMAN: Robotin ajurijärjestelmän modularisointi liitännäisarkkitehtuurilla
Tampereen Teknillinen Yliopisto
Diplomityö, 42 sivua
12. syyskuuta 2018
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotekniikka
Tarkastaja: Asst. Prof. Outi Sievi-Korte

Avainsanat: Modulaarisuusanalyysi, Robotiikka, Liitännäisarkkitehtuuri, Python

Tehdaslinjastot koostuvat nykyään älykkäistä järjestelmistä, jotka kykenevät toimimaan ilman ihmisen ohjausta. Merkittävä tekijä tämän muutoksen takana on ohjelmistojen kehittyminen, ja niiden merkityksen kasvu teollisuudessa. Tämä luo ohjelmistosuunnitteluun uusia haasteita, joita on ratkottu uusilla sovellusalustoilla ja -kehyksillä.

Tässä tutkielmassa toteutetaan uusi ohjelmistoarkkitehtuuri testiautomaattirobotille käyttäen liitännäisarkkitehtuuria. Tavoitteena on kasvattaa alustan modulaarisuutta, mikä mahdollistaa sen joustavan käytön erilaisissa järjestelmissä, jotka koostuvat erilaisista roboteista, antureista ja sensoreista.

Uuden toteutuksen soveltuvuuden varmistamiseksi tässä tutkielmassa suoritetaan modulaarisuusanalyysi molemmille järjestelmille. Analyysissä perehdytään järjestelmien luokkien ja moduulien yhteenkuuluvuuteen (eng. cohesion) sekä riippuvuuksiin (eng. coupling). Järjestelmä toteutetaan Python-kielellä, joten tutkimuksessa selvitetään modulaarisuusanalyysin soveltuvuutta dynaamisesti tyypitetylle ohjelmointikielelle.

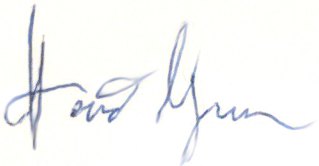
Tutkimuksessa todettiin luokkien yhteenkuuluvuuden kasvaneen ja moduulien riippuvuuksien vähentyneen, mikä kertoo järjestelmän modulaarisuuden kasvusta. Analyysi itsessään todettiin työlääksi, ja sen huomattiin kasvattavan tulosten virhettä, koska moduulien väliset vuorovaikutukset eivät välttämättä olleet yksikäsitteisiä. Tutkimuksessa pohdittiin mahdollisuudesta muokata olemassa olevia refaktorointityökaluja siten, että niitä voisi hyödyntää tämänkaltaisen analyysin tekemisessä.

PREFACE

I want to thank my instructor Outi Sievi-Korte for guiding and supporting me through this challenge, as well as Hans Kuosmanen and Tapio Marttila from OptoFidelity for offering me this interesting subject. A huge thanks goes also to my supervisor Nikhil Pachhandara for supporting me and helping me allocate time for my thesis work.

I would also like to thank my wife for tolerating me while I have had my face buried in the laptop screen for half a year. The greatest of thanks goes to our cat who never failed to comfort me during the toughest of times.

In Tampere, Finland, on September 12, 2018

A handwritten signature in blue ink, appearing to read "Henrik Nyman", written on a light yellow background.

Henrik Nyman

CONTENTS

1. Introduction	1
2. Software architectures and modularity	4
2.1 Software architectures	4
2.2 Measuring software modularity	5
2.2.1 Cohesion	5
2.2.2 Coupling	6
2.2.3 Reconfigurability	8
2.2.4 Modularity in a dynamic language	9
2.3 Plugin architecture	9
2.4 Plugins in Python	12
3. Method and context	15
3.1 A case study	15
3.2 Industrial robotics	16
3.3 Current system	17
3.3.1 Architecture	18
3.3.2 Weaknesses	21
4. Architecture design of Test Factory	24
4.1 Core	24
4.2 Plugin system	25
4.3 Message bus	27
4.4 Driver interface	28
4.5 Configuration	30
4.6 Logging	32
5. Evaluation	34
5.1 Cohesion	34
5.2 Coupling	36

5.3 Reconfigurability	39
6. Conclusion	40
6.1 Validity of the study	41
6.2 Further research	41
References	43

LIST OF FIGURES

1.1	Basic overview of the sample system. (Images: Alexander Lucke 2011. SVS-VISTEK - SVCam-ECO BlackLine - Tubus. From: [19]. Google Android 2013. Photo of the Nexus 5. From [11]. Jo Teichmann, Augsburg, Germany (KUKA Roboter GmbH) 2011. KUKA industrial robot arm. From: [11])	2
2.1	Eclipse extension framework. Adapted from [2].	11
3.1	OptoFidelity 2018. Fusion Tester. From: Fusion product information, OptoFidelity website. [24].	17
3.2	OptoFidelity 2018. Test Factory concept image. Test Factory product information, OptoFidelity website. [25].	18
3.3	Component diagram of Fusion software architecture	19
3.4	Class diagram of Fusion Sequencer	20
3.5	State diagram of Fusion Sequencer's state machine	21
4.1	Component diagram of Test Factory software architecture	24
4.2	Class diagram of Test Factory Core	26
4.3	Test Factory messaging interface	27
4.4	Sequence diagram of the startup of Test Factory	30
4.5	Capturing three images on Fusion	31
4.6	Capturing three images on Test Factory	31
5.1	Histograms describing the distribution of coupling values of the target systems. The blue bar represents direct coupling, and the red bar indirect coupling.	38

LIST OF TABLES

2.1	Levels of coupling [13]	7
4.1	Websocket message types	28
5.1	Cohesion of Fusion	35
5.2	Cohesion of Test Factory	35
5.3	Coupling values of Fusion	36
5.4	Number of interconnections in Fusion	36
5.5	Coupling values of Test Factory	37
5.6	Number of interconnections in Test Factory	37
6.1	Summary of the research.	40

LIST OF ABBREVIATIONS AND SYMBOLS

API	<i>Application Programming Interface.</i>
DUT	<i>Device Under Test.</i> Abbreviation for a manufactured product that is the target of a functional test.
GUI	<i>Graphical User Interface.</i>
I/O	<i>Input/Output,</i> a piece of hardware used to communicate with a computer either by human or another machine.
IDE	<i>Integrated Development Environment.</i> A software that usually bundles together a text editor, compiler and a debugger for a given language.
Industry 4.0	The fourth industrial revolution, factory automation and big data analysis. [33]
IoT	<i>Internet of Things.</i> A networked interconnection of everyday objects which are often equipped with ubiquitous intelligence. [34]
RESTful API	A programming interface that uses HTTP requests to manage data. Typically GET, POST, PUT and DELETE are used.
TCP	<i>Transmission Control Protocol.</i>
TnT	OptoFidelity <i>Touch and Test</i> platform

1. INTRODUCTION

While Internet of Things (IoT) is revolutionizing many consumer product families, Industrial Internet of Things (IIoT) and Industry 4.0 are doing the same for factory equipment [33]. Factory line machines are no longer independent and isolated, but instead connected to a factory network through which they can be controlled and monitored.

The impact of Industry 4.0 is already huge. It is estimated that it drives the growth in manufacturing in Germany alone €30 billion each year. It is also to increase productivity in German manufacturing between 5 – 8% annually, one of the biggest beneficiaries being the automotive industry [27].

Robots are a central component of any smart factory layout. They can be the manufacturer of the factory line, in which case the robot is doing the main work performed in the line, or they can act as transport devices, moving the products from one manufacturing step to another. A completely autonomous factory line is such where both of these tasks are performed by a robot. A factory that is completely operated by robots is also known as a “dark factory”, named by its lack of need of lights to operate.

When robots get more complex, demands for their software get more intense. A simple industrial robot can perform its task with a very simple software controller. Perhaps it performs only a single task, and thus may require only one procedure bundled with a few error condition handlers. However, when the process is no longer simple and the hardware contains many varying parts, it becomes increasingly difficult to expose different types of components in a way that is homogenous enough while still preserving enough control.

This thesis describes a research project that redesigns the software architecture of a robotics system made for testing smartphones. The system consists of one robot, which may have several axes, and multiple peripherals such as cameras, sensors, microphones and speakers. Those peripherals may be attached to robot’s axis — in which case they move along with the robot — or they can be stationary. Software for these systems needs to be modular and configurable, because these robots are manufactured in different setups containing varying set of peripherals. An example system with one camera as a peripheral can be seen in figure 1.1.

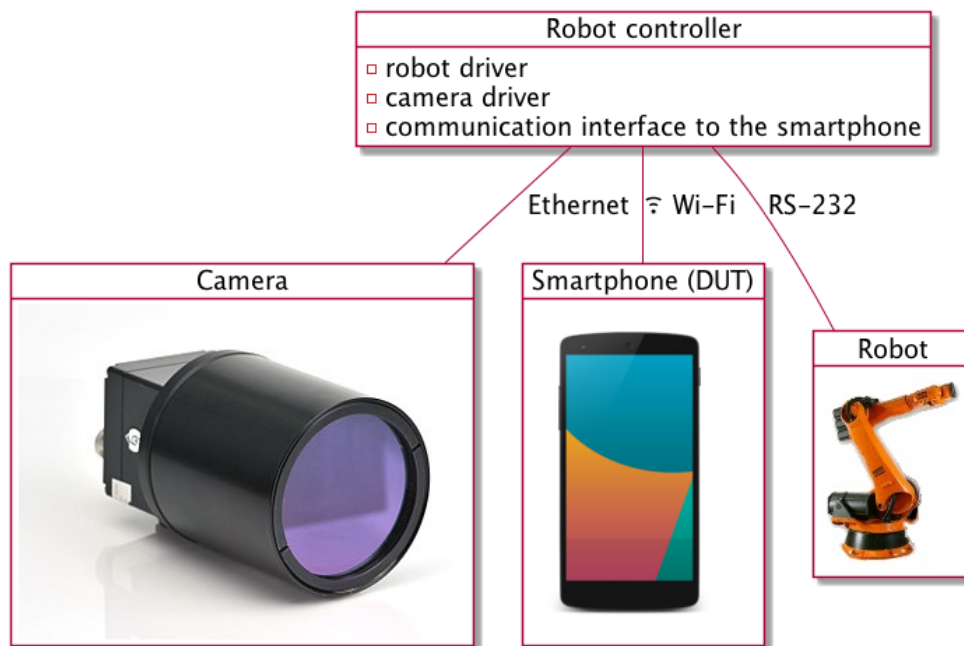


Figure 1.1 Basic overview of the sample system. (Images: Alexander Lucke 2011. SVS-VISTEK - SVCam-ECO BlackLine - Tubus. From: [19]. Google Android 2013. Photo of the Nexus 5. From [11]. Jo Teichmann, Augsburg, Germany (KUKA Roboter GmbH) 2011. KUKA industrial robot arm. From: [11])

The research focuses on the architecture design from the software perspective, and does not cover implementation details or hardware. Low-level control software for the robot and its peripherals is only covered within the interface they provide via their plugin. System's user interface is also outside of the scope of this thesis. The test application running on the smartphone is presented in the architecture diagrams, but its internals are not covered in this thesis.

The thesis aims to answer to following research question: *How to increase modularity and configurability of a robotics system?* A subproblem of this is to evaluate how well does modularity analysis carry out with Python – a dynamically typed language – compared to statically typed languages on which these analyses are normally conducted on.

The research compares the software architectures of the current and new systems from the context of modularity and configurability. It describes the main pain points of the old architecture and how the new architecture aims to asses them. The new architecture is designed to fit more use cases than the old one, but concerning this thesis the architectural redesign is done with the current hardware setup in mind.

Structure of the thesis is as follows. The second chapter gives background in general about robotics systems and modularized architectures. Chapter 3 presents the current software architecture and defines it as the context of this study. In chapter 4 the new architecture

design is explained, and the chapter after that tells how the architecture is applied in deploying the system. Finally, in chapters 5 and 6 the architecture is analyzed with methods defined in chapter 2 and concluded with a discussion on the successfulness of the method and thoughts on further research.

2. SOFTWARE ARCHITECTURES AND MODULARITY

This chapter creates a basis on the further discussion on the topic of software architectures and modularity. Firstly, the concepts are defined in a more general level, and later with Python and dynamic programming languages in mind. This chapter also presents the methods used in the evaluation in chapter 5, and explains why those analyses were selected.

2.1 Software architectures

In construction an architecture refers to the way a building is designed in the highest level. In software engineering, architecture describes the exact same thing: how components are laid out in the highest level, how they relate to each other and how they communicate. Bass, Clements and Kazman [1] define a software architecture as a set of software structures that are held together by a relation, such as association, inheritance, or an abstraction from the real world.

The main purpose of software architecture is to make a large piece of software easier to manage and maintain. It aims to split the system components into groups which are tied together by a semantical meaning. There are many approaches than are used commonly, for example in a client-server -pattern the functionality is distributed in a way that multiple clients can ask the server to perform work, whereas the server defines with its API what kind of operations it provides for the clients. On the other hand, in MVC (Model-View-Controller) -model the application is divided into the data and operations (model), user interface and/or API (view), and handling of the input from the user or the client using the API (controller).

Bass, Clements and Kazman [1] also discuss what are the building blocks of a good software architecture. Their extensive list can be boiled down to four rules of thumb:

1. Value the quality attributes over the feature set

2. Make modules well-defined and self-contained, make components interact in a predictable way
3. Stick to well-known patterns if at all possible
4. Have the high-level architecture designed by a single person

They claim that by following forementioned principles will make it more likely for the software to fulfil its requirements both from a technical standpoint and from the business perspective.

2.2 Measuring software modularity

The general consensus is that a good principle in software design is to have module cohesion as high as possible, and keep module coupling to the minimum [30, 14, 20, 4, 3], as it leads to logical structures and few dependencies between parts that are not directly related. Following these fundamentals results in naturally modular software components: their modules contain only functions and attributes that have something in common, and the modules depend only on the public interface of other modules.

Plenty of research has been done on how to measure software modularity. Dörbecker, Böhm and Böhm have listed in their literature review [6] a large amount of papers that do modularity analysis on some level. They categorize the papers based on what effects of modularization they target. In the context of software most of the sources focused on cohesion, commonality and coupling, and some additionally on redesign or reconfiguration. In this paper we focus on measuring only *cohesion* and *coupling*, as those are the most common properties measured in modularity analyses, based on [6]. Chidamber and Kemerer [3] also define cohesion and coupling as two important design concepts of object-oriented software design. The reconfigurability analysis is also discussed briefly.

In the following subchapters present the selected methods for the analysis. The choice of methods was done by referring to [6], and evaluating how well the characteristics required in the calculations are available in the target systems. In the end based on that criteria, [12] for cohesion and [13] for coupling were selected.

2.2.1 Cohesion

Generally cohesion describes how units form a whole. In software engineering context cohesion tells how well functions in a module belong together, and how little there exists

functionality which does not logically need to be in the module. A high functional cohesion exists when all the elements of a class work together to provide some *well-bound* behaviour. [10]

Chidamber and Kemerer [3] present a metrics suite for analyzing object-oriented software designs, and define a metric for both cohesion and coupling. Of those, the method for measuring cohesion is utilized in this research work. A similar method for cohesion measurement is introduced in the article [12] written by Gui and Scott. They define the cohesion of a class to be based on the similarity of the methods, the similarity being a measure of how much the sets of the instance variables the classes access overlap. First, we let $M \equiv \{M_1, M_2, \dots, M_m\}$ be the methods of a class C , and $V_j \equiv \{V_{j,1}, V_{j,2}, \dots, V_{j,n}\}$ instance variables that method M_j accesses. The direct similarity $SimD$ between two methods can be calculated with

$$SimD(i, j) = \frac{|V_i \cap V_j|}{|V_i \cup V_j|}. \quad (2.1)$$

The resulting value falls inside range $[0, 1]$. Gui and Scott extend the measure to take indirect similarity into account, but in the context of this thesis, only direct cohesion is measured. From the pairwise similarities of all the methods in the class we get to the cohesion score $ClassCoh$ of the whole class:

$$ClassCoh(C) = \frac{\sum_{i,j=1}^m Sim(i, j)}{m^2 - m}. \quad (2.2)$$

Furthermore, we can calculate the cohesion of the whole application or any subset of its classes with:

$$WTCoh = \frac{\sum_{j=1}^n ClassCoh(C_j)}{n}, \quad (2.3)$$

where we define $C \equiv \{C_1, C_2, \dots, C_n\}$ as the n classes of the system.

2.2.2 Coupling

Two objects have coupling if and only if one of them affects the state history of the other, where the actions performed results in the other object to be in a different state than it

would be without the action. In a highly modular system software components have a loose coupling between each other. Harrold and Kolte present a quantified method for analysing module coupling [13].

Harrold and Kolte used the method to analyze C code, but in this thesis we apply it to a Python source. Due to Python's dynamic nature, it is more difficult to obtain information on bindings between modules. In C, the information can be extracted with a compiler or even parsing the source code with regular expressions, but in Python we cannot get ourselves to that information especially if there are no type annotations. A dynamic code analysis tool such as Jedi [5] can be used to generate data by deducing references from annotations and constants. However, it does not have a built-in support for this, and in the end some of the work needs to be done manually anyway.

Myers [21] defines six levels of coupling. Harrold and Kolte add a level zero to indicate completely independent modules. All coupling levels and their definitions are listed in table 2.1. If a module is found to be coupled in more than one level, the level of coupling for that module is the highest of them. Some of the level descriptions are adjusted to better fit Python paradigms.

Table 2.1 Levels of coupling [13]

Level	Name	Description
0	Independent coupling	No coupling between the modules.
1	Data coupling	The modules pass data via function or class parameters.
2	Stamp coupling	The modules pass data via reference parameters.
3	Control coupling	The modules pass a flag parameter that alters modules behaviour.
4	External coupling	The modules communicate through an external medium such as file or a database.
5	Common coupling	The modules refer to same global data.
6	Content coupling	The modules access and change each others internal data.

To calculate the coupling between two modules x and y , as presented by Fenton and Melton [7]:

$$M(x, y) = i + \frac{n}{n + 1}, \quad (2.4)$$

where i is the highest level of the coupling between the modules and n is the number of interconnections between them. Harrold and Kolte adjust the equation to make M fall into

interval $[i, i + \frac{1}{2}]$ by subtracting $\frac{1}{2}$ from it:

$$M(x, y) = i + \frac{n}{n+1} - \frac{1}{2} = i + \frac{n-1}{2(n+1)}. \quad (2.5)$$

To further get to the coupling value of the whole application (or a subset of the application modules in this paper’s case) we need to calculate the coupling scores for all modules pairwise. [7] suggests using the median value of those scores as the total coupling of the system, but we take the approach of [12]; calculating the weighted transitive coupling *WTCoup* of the system:

$$WTCoup = \frac{\sum_{i,j=1}^n M(i, j)}{n^2 - n}, \quad (2.6)$$

as was done when calculating the cohesion score of a single class in the previous section.

The numerical values extracted from the analysis do not tell the whole truth, though. The analysis does not reveal a “hidden” coupling, where a module depends on another module in a semantical level: when modifying one module, the other needs to have corresponding modifications as well. This phenomena is analyzed manually in the chapter 5.

2.2.3 Reconfigurability

Reconfigurability refers to an ability to use the same software components to build a variety of different systems. This feature can be implemented in different ways, such as using preprocessor directives in C-language to build different binaries from the C-code, or parametrizing Python classes during runtime with a configuration file. A common aspect for both is that the source code is not modified in order to produce the artifact variants. [18]

Liebig et al. [18] study configurability in the context of refactoring engines. They focus on how different kind of refactoring engines handle — or do not handle — configurability implemented with C preprocessor directives. They tested 18 different refactoring engines and discovered that 7 of them supported no proper refactoring support when static reconfigurability is present. Thus, even with a statically typed, compiled language, increasing configurability reduces the effectiveness of static analysis on the code base.

In this research the configurability of a system is measured on how many different hard-

ware configurations can be supported with no modifications to the source code. As discovered in [18], even a few preprocessor options increase the amount of different software configurations quickly by a huge amount. Using a configuration file is no different, and it does not make sense to compare whether the application has two million or two hundred million different variants. That is why in this thesis instead of measuring the amount of unique configurations, we measure the amount of configurability points; places in the system where the configuration affects the way the software behaves.

2.2.4 Modularity in a dynamic language

The dynamic nature of Python affects some aspects of measuring coupling. Differentiating between levels 1 and 2 is not as straight forward as it is in C, since Python does not have a syntactic way to define pass-by-value or pass-by-reference. Instead, the type of the parameter given to the function deduces the parameter passing style: immutable types such as `int`, `float`, `bool` and `tuple` get passed by value, and mutable types including `list` and `dict` get passed by reference. To pass object types by value the programmer needs to make a copy of the object manually, which can be done either in the caller or the callee side. For this reason the parameter passing style cannot be determined from a bare function declaration as it can be done with C code.

Python has no real concept of private attributes. Typically private members are indicated by defining them with a name starting with an underscore. The language, however, does not do anything to prevent a programmer from accessing the “private” member from outside of the class. To make the member even more private, the programmer can use a name which starts with two underscores¹. In this case Python triggers name mangling on that variable and injects the name of the class into the attribute name if it is accessed outside of the class. Again, if the programmer knows the name of the class, this attribute can be normally accessed from the outside. [8] For this reason it may be difficult to define when a module accesses private members of another module.

2.3 Plugin architecture

Plugin architecture is one of the ways of implementing a modular system. The main goal of a plugin is to provide a way to add a feature, or modify an existing feature of an application without modifying or recompiling the application itself, and to reach modularity by

¹But which does not end with two underscores. Names starting and ending with double-underscores are reserved names in Python. Again, the language does nothing to prevent naming variables with this style, but it is considered a bad practice. [8]

spreading separate functionalities into entirely separate components. Whereas traditional methods for modularization such as object-orientation, procedures and components work in the source code level, plugins reach the modularity at application level. The two are, however, not exclusive and combining them results in software that is modular on multiple levels. Obviously using a plugin architecture does not magically fix lower-level issues in the code base or vice-versa.

The main driver for selecting plugin architecture as the design of choice in the project was that it allows spreading the implementation details not only inside the team but inside the company as well. It was considered superior to other options. Let us first consider implementing the internals of the driver as a separate library. It would encapsulate the implementation details well, and would be easy to allocate a separate team for its development. However, to use it with the core application, a binding code needs to be written. The binding code would not be suitable to be written inside the library, as it is not generic. If the binding code would be added to the host application, it would make the core dependent on a specific version of the library.

Another option would have been to implement the binding code as a separate controller module. This is somewhat similar to the approach used currently in TnT Server, and it lacks the ease of spreading the implementation of the modules to the teams that have the greatest expertise on that specific subdomain.

Wagner et al. [32] list benefits of plugin architecture in heuristic optimization software systems. These benefits include reduction of *complexity*, increase of *configurability*, *customizability*, and *expendability*, and simplification of *deployment*. In the paper, they conclude that the biggest reason plugin architecture has not raised its head in heuristic optimization systems is that it is really difficult to create a common *meta-model* for that kind of tasks. By *meta-model* they refer to characteristics that unify common aspects of different heuristic algorithms that can be isolated to form separate packages connected with interfaces. Therefore, not all software is easily spread into plugins; it may require redesigning the core concept of the system, or it may not be feasible at all.

The core application defines what functionality it exposes to its plugins by providing them an API. This API can consist of functions that query the state of the core program, and “hooks” that allow the plugin to react to certain events occurring in the core application. The core can also provide means for the plugin to create and manage their own user interface elements such as toolbar buttons or context menu items. The amount of flexibility the core allows can vary a lot: from offering a very limited set of APIs, or exposing all of the internals for plugins to use. Coming up with a good plugin architecture is about finding a balance between offered functionality and complexity of handling plugins’ interactions

with core and with each other. [15]

An example implementation of a plugin system is shown in figure 2.1, which illustrates a context-menu action plugin for the popular open-source IDE Eclipse. In the figure, a plugin binds to the `actionSets`-interface to provide additional actions `HelpContentsAction` and `OpenHelpSearchPageAction`. The plugin does not know – or care – how the user interface elements are implemented, and the core does not need to know what functionality is attached to the menu items, it just calls the action’s callback method when selected by the user.

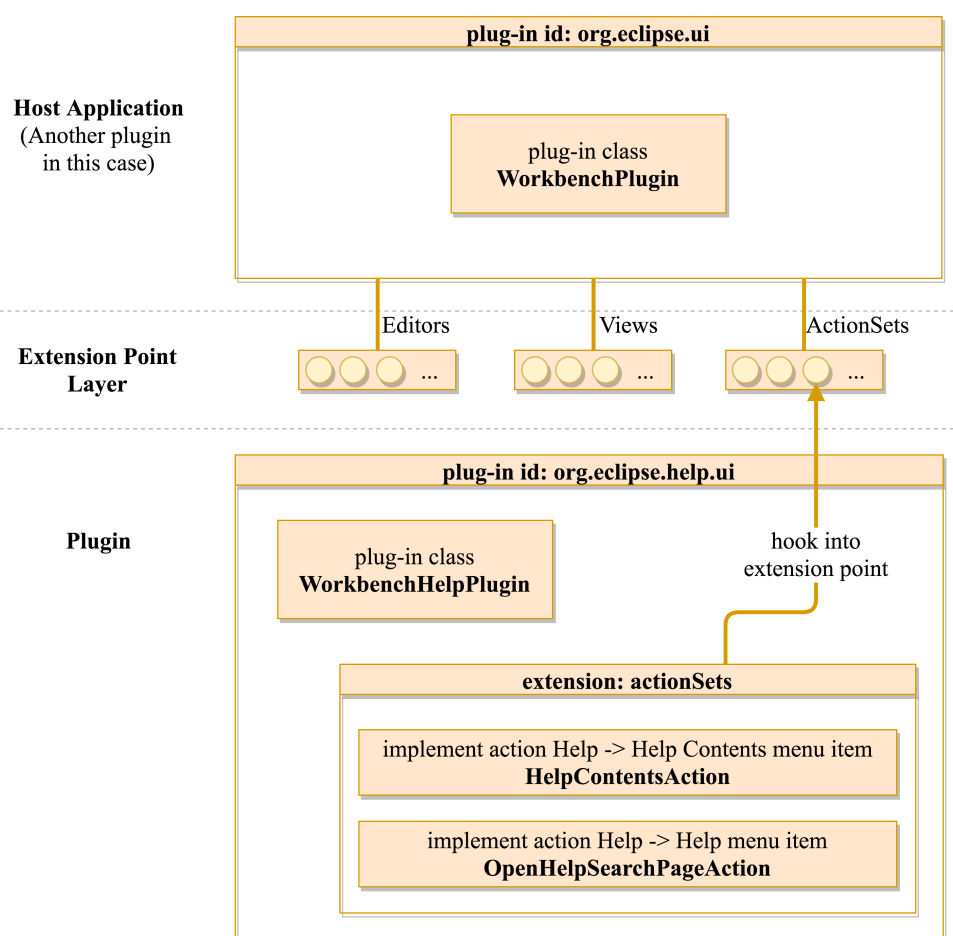


Figure 2.1 Eclipse extension framework. Adapted from [2].

Plugins allow for more flexibility in software business, too. By having features as separate packages that are released individually from the core application, it is trivial to implement license handling for individual features. The author can easily sell the same application with different feature sets to different customers with different pricing models. The author can also allow for third parties to create plugins for the application, and build business models around that use case as well. [15]

Plugin APIs are analogous to other kinds of software APIs when it comes to changing

the interface. Adding functionality is always safe, but changing the behaviour of existing methods or even removing them altogether will almost certainly cause issues with plugins using that functionality. This is what happened with Firefox in late 2017 when they moved into a completely new plugin API, which was way more restrictive than the old one.² Original API was exceptionally flexible, giving the plugins access to the internals of the browser which enabled developing rich and powerful plugins. The new API is more restrictive and tries to be similar to the extension API provided by other browsers, to allow for easier development of cross-browser extensions.

2.4 Plugins in Python

Plugin architecture can be implemented in any programming language. However, this thesis project is implemented in Python, so implementing the architecture in Python is discussed more thoroughly. This subchapter describes what methods Python offers to implement a plugin-based system, and what kind of method is selected to be implemented in the project.

Python being an interpreted language, creating a plugin is really flexible. In its simplest form, a plugin is just a regular source file which for example resides in a directory known to the core application, from where it can directly import it, and call its functions. This has a downside, though, as the file needs to be in a known location, which may be infeasible if the modules are installed with `setuptools`.

In this thesis project, a plugin-API based on entry points defined using the `pkg_resources`-module. This method outsources the dynamic loading of the extension code to an external module, and makes loading plugins possible without knowing where the actual plugin is located on disk or what module implements it.

Exposing a method or a class from a package is done in the `setup.py`-file with an entry point declaration. An entry point declaration consists of three parameters: *group*, *name* and *package*. The *group* defines the purpose of this entry point. There are built-in groups such as `console_scripts` for implementing shell commands, but in case of a plugin we can just choose a descriptive group name. The *name* identifies this entry point inside its group. The same package may provide multiple entry points with the same group but a different name. The *package* tells Python what file and class/function in the module implements this entry point. An example of an entry point definition is shown in listing 2.1.

²Mozilla announced in September 2015 [23] that Firefox will deprecate XUL and XPCOM interfaces and force developer to use WebExtension framework to build plugins, resulting in multiple extensions using the old APIs to stop working. These changes were released in version 57.0 in November 2017 [22, 31].

The module can expose the entry point even if there is no application installed that uses those entries. This is useful when the module contains other functionality in addition to the plugin part, allowing a module to expose its functionality as a plugin while still allowing for stand-alone use without the plugin host.

```

from setuptools import setup

setup(name='sample-package',
      ...
      entry_points={

          # This plugin is implemented in sample_package/plugin.py
          'plugin_framework.plugin': [
              'my_plugin = sample_package.plugin:MyPlugin'
          ]
      },
      ...)

```

Listing 2.1 Entry point definition in setup.py-file

To load the extension from the main application, pkg_resources-module is utilized again. Example code loading the plugins defined in the previous example is written in listing 2.2. The example loads all the entry points defined in all installed python packages. It is also possible to load a plugin directly by its group and name.

```

from pkg_resources import iter_entry_points

for entry_point in iter_entry_points('plugin_framework.plugin'):

    # driver_name will be 'my_plugin'
    driver_name = entry_point.name
    cls = entry_point.load()
    driver = cls()

```

Listing 2.2 Code loading all entry points defined with the group name.

Definition of the API can be implemented in multiple ways. Since the method discovery does not require any special declaration in the module itself, the simplest way to achieve a defined API is to *document* what function names the module should contain and what are their arguments. The downside for this is that the documentation becomes easily out-of-date, and that it is impossible for an IDE to provide for example auto-completion of method names and arguments for the plugin. This project takes the subclassing approach: the core application defines a base class for all of the plugins to specialize. This creates a two-way link between the core and the plugin, and allows the core to easily define default behaviour

for plugins that do not implement a specific aspect. Subclassing method requires the plugin to import the base class from the core, so the programmer needs to be careful that the base class can always be imported with the same module name between updates, and that importing the class does not cause unexpected side-effects.

3. METHOD AND CONTEXT

This chapter defines the research method and describes the starting point for this research. A case study is described in software engineering context, and the current architecture is illustrated as the context of the study. Then we introduce the field of industrial robotics — the context of this project, and describe how it creates an environment which differs from traditional software engineering research environments. In the end we give an overview of the robotics system currently in use and its software architecture, and discusses its weaknesses.

3.1 A case study

Also in software engineering context, a case study is most often conducted as a qualitative research, as stated by Runeson and Höst in their paper on case study guidelines in the field of software engineering [26, p. 135]. According to the article in an exploratory case study, insights to future research are an important side aspect along with the direct results from the study. This thesis work implements a qualitative research with an exploratory purpose. Measurements of software modularity have been focusing on statically typed languages. This research aims to generate ideas for more thorough modularity analysis for Python as well as for other dynamic languages.

Case study was a clear method of choice for this research. The redesign project fits as the subject on this study, while representing a single instance of modularization analysis. The subject is also fairly complex, which allows for diving into smaller details which is typical in a case study.

The project was started in the first half of year of 2018 by beginning the architecture redesign work. A while later the project was found to be suitable for a modularity analysis, since one of its targets was to increase the modularity of the system. Therefore, it was selected as the case for the study. The research work was started in June 2018 and finished by the fall 2018.

3.2 Industrial robotics

Mechanical appliances have aided humans in many tasks that require repetition, precision or strength. To further reduce the role a human plays in manufacturing and quality-control, robots have started to take over the industry. The human has retreated from the front line of the factory to the base where he designs and builds the automation systems doing the actual work. The mechanical side is still a crucial aspect of an automation system, but more and more of the work is targeted at the software controller and firmware of the system.

A term industrial robot does not necessarily always mean a huge robot arm capable of lifting car parts. It can as well be a simple actuator, or an automated conveyor belt. The common feature is always that instead of being controlled directly by a human, it is commanded by its software controller, or it is a part of a bigger system. These robots can then be connected to a monitoring system and to a data gathering system which provides automated telemetry logging, diagnostics and performance analyses.

Industrial robots are spreading The prices of industrial robots have dropped significantly during the last few years, which has made them available to smaller scale manufacturers, too, as it does not require a huge production throughput to overcome the expenses. As the complexity moves from the mechanical side to the software side, it becomes possible to utilize the same robot in multiple different use cases. [28]

This thesis focuses on test automation, which is a subdomain of industrial robotics. It aims to replace humans responsible for quality-control with an objective qualification machine that produces reproducible results. Its software needs to reach a high quality in both data gathering and data analysis. The environment, being a factory line, it is often filled with loud noises and dust particles that cause additional distortion to the collected data [17]. To overcome that, the system has to either implement filters to reduce the interference or compensate for them in software, latter of which is usually the cheaper solution. It, however, creates additional requirements for the software being able to adapt to the hardware and the environment.

Software architectures in the area of robotics have been studied before. García et al. [9] discuss modular software architectures in robotics systems. They present a hardware configuration that is similar to the one discussed in this thesis, followed by a set of targets they try to achieve with their architecture. The robot they are using is a robot arm with six degrees of freedom, and it is designed to be equipped with a variety of different sensors and controllers. The new architecture was seen to improve the support for having plug-and-play components in the system.

3.3 Current system

The context of this thesis is the software architecture of OptoFidelity™ Fusion Tester. It is an industrial robot designed for smartphone functional testing covering buttons, connectors, display, speakers, cameras, microphones and sensors. It is targeted at usage as a factory line final tester or as an all-in-one test system in refurbishment business. A Fusion robot is presented in figure 3.1.



Figure 3.1 OptoFidelity 2018. Fusion Tester. From: Fusion product information, OptoFidelity website. [24].

The main selling points of the robot are its high test coverage and fast cycle time. It aims to replace the human operator in smartphone functional testing with a fast and robust robotics solution. Traditionally smartphone testers use device-specific mechanical adapters that handle the differences between different DUTs (Device Under Test), but Fusion takes a different approach. It holds onto the phone with a suction cup and handles differences between DUTs by defining offsets in a DUT configuration file and using those offsets when operating the device with the robot. The device placed inside the robot is recognized with a special QR Code on the device's display ¹.

As the successor of Fusion a product called Test Factory is being developed. Test Factory

¹The QR code contains the model name of the device, and IP/port-combination for establishing the communication link to the device. From the position and orientation of the QR code the robot can determine the position and orientation of the device in robot's coordinate frame. [16]

is hardware-wise similar to Fusion except that it does not necessarily aim to be an all-in-one tester, but rather a modular test framework, where individual robots can specialize into different testing capabilities. This way the total throughput of the factory line can be increased by pipelining [29] together multiple robots with shorter cycle times, each doing their own part of the whole test run. A concept picture of Test Factory system can be seen in figure 3.2.



Figure 3.2 OptoFidelity 2018. Test Factory concept image. Test Factory product information, OptoFidelity website. [25].

Software architecture of Fusion does not directly fit into this use case, which resulted in this research work. Furthermore, with a successful redesign of the architecture, both systems would be able to use the same code base, which would streamline future development of both products.

3.3.1 Architecture

Software architecture of Fusion is divided in the highest level into three parts: *TnT Server*, *sequencer* and the *dut application* running in the smartphone. The main software components of Fusion tester are laid out in figure 3.3.

TnT Server is the part which directly interacts with hardware. It is used by multiple projects in the company, so it is designed to be very generic. It contains drivers and

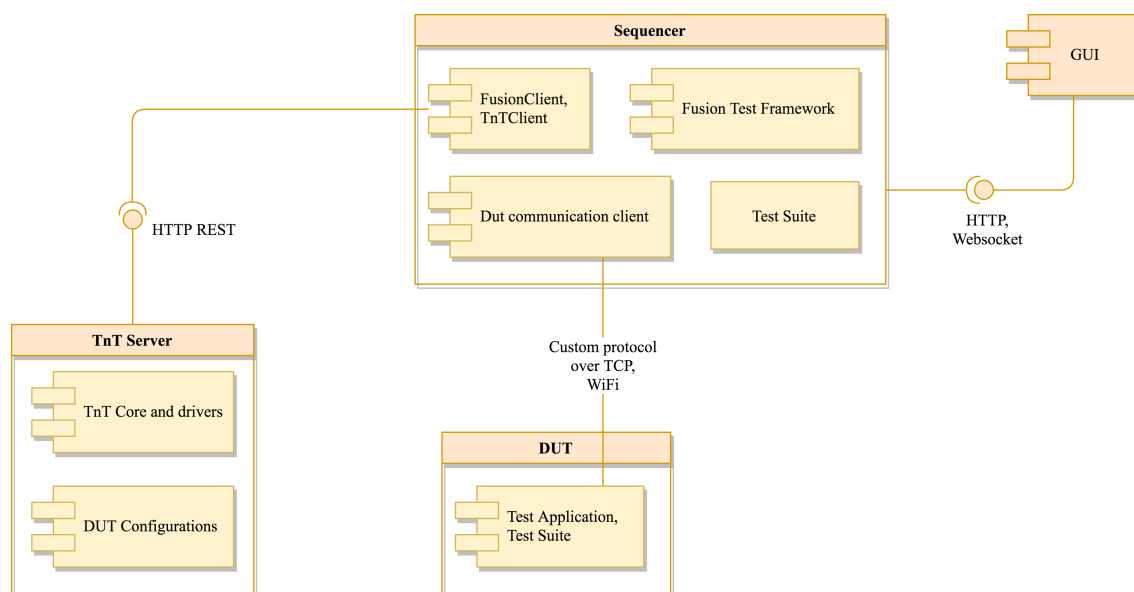


Figure 3.3 Component diagram of Fusion software architecture

controllers for all of the hardware components in the system, and exposes them through a RESTful API to external components. To utilize the REST API from Python, TnT Server provides a client library that can be imported in the controlling application.

TnT Server has been designed rather modularly, by dividing the application into *resources* and *controllers*; following sort of a model-controller design pattern. A resource is practically a device driver. It implements a set of methods for the device, which can then be used by the controllers. A resource can be a subclass of another resource and implement the same methods, in which case both can be used with the same controllers. The controller defines what features of the whole application are exposed through the REST API, and how the features are laid out on the URL dispatcher. The features exposed by the controller can be on a higher abstraction level than the ones implemented by the driver, or it can merely pass the requests directly to the driver to provide a low-level control.

Initialization procedures for all hardware components are executed in TnT Server. If there is a faulty or a misconfigured component, the REST interface will not start, and the sequencer will go into corresponding state.

The sequencer ties together TnT Server, a graphical user interface and the DUT application controller, and binds them into a high-level state machine. It handles the initialization the test cycle: moving the robot to the start position, waiting until a DUT is placed on the holder and establishing a TCP (Transmission Control Protocol) connection with the device. Then it loads the test set configured for that device and starts executing the cases one by one. A class diagram of Fusion sequencer is shown in figure 3.4.

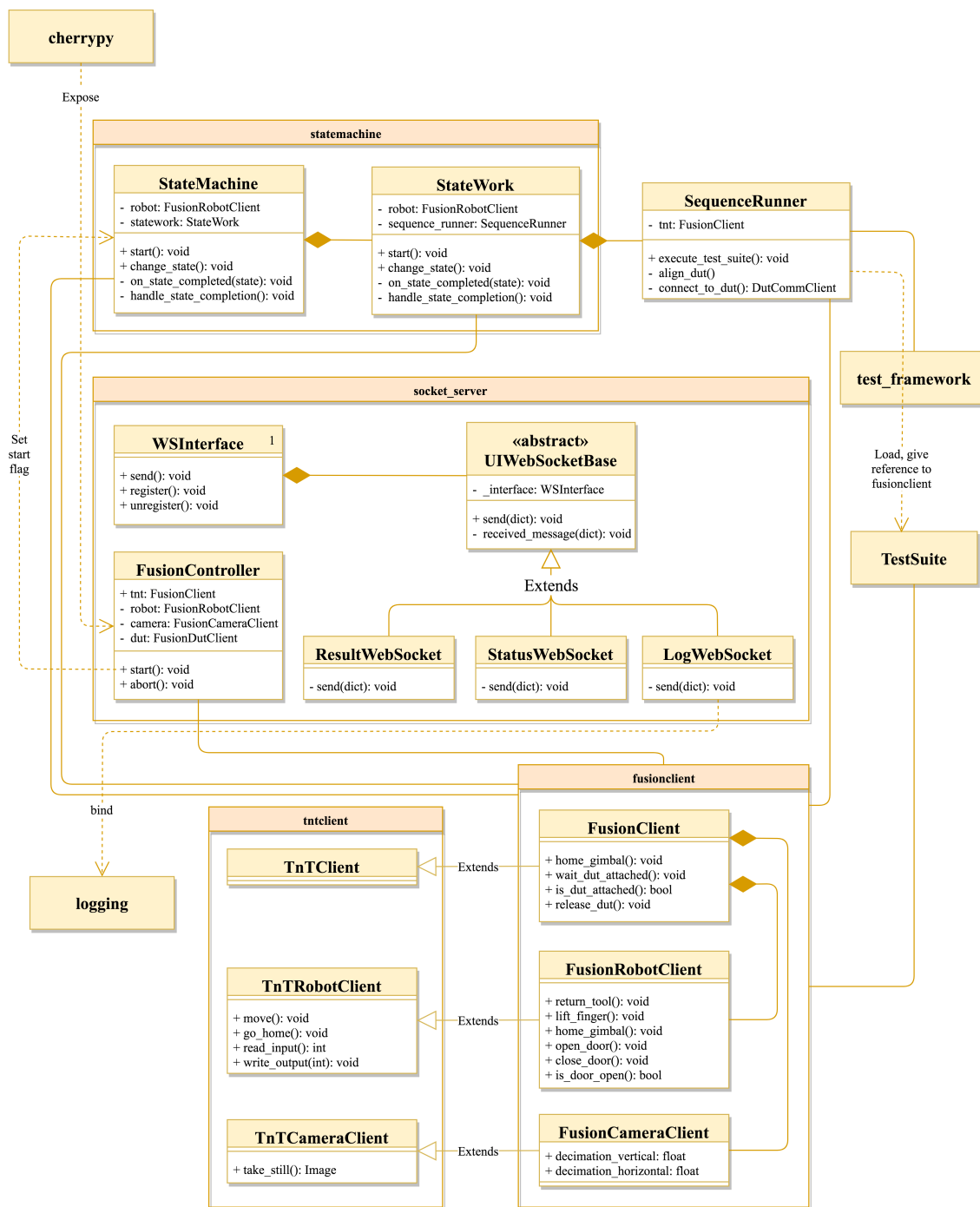


Figure 3.4 Class diagram of Fusion Sequencer

In the ideal case, the state machine iterates the sequencer between two states: *Wait_StartButton* and *Executing_Sequence*. If there are no exceptional conditions, the sequencer never goes outside these two states. The state machine is a high-level state machine and its main purpose is to prevent running two test suites simultaneously and to handle robot errors and emergency stop presses. The state diagram of the state machine is shown in figure 3.5.

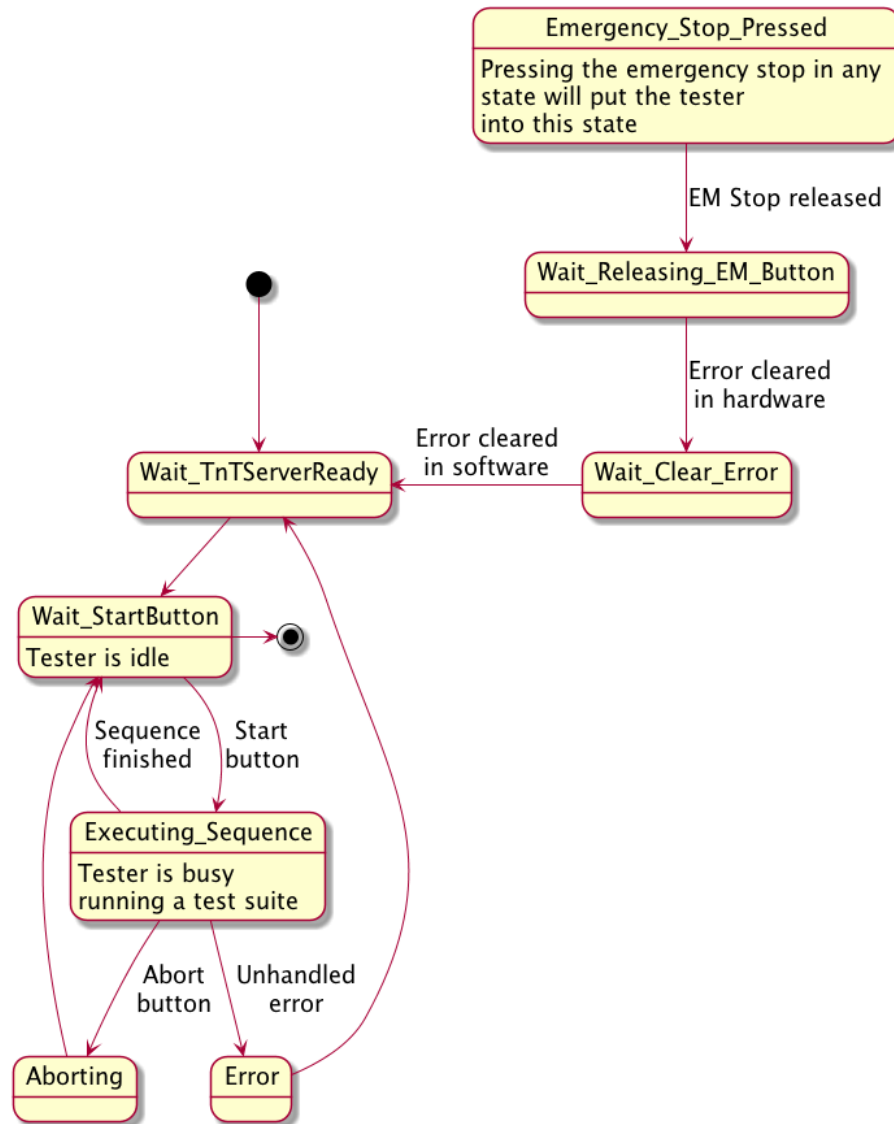


Figure 3.5 State diagram of Fusion Sequencer's state machine

3.3.2 Weaknesses

This section describes the issues the current architecture has more or less related to modularity. It tries to delve into the reasons why these decisions were made, and how they affect the development of the modules now and in the future.

Most issues regarding to modularity in Fusion root down to the tight coupling of TnT drivers with TnT Server core and therefore also with TnT Client. When support for a new device is added, changes are needed in several modules:

1. A new resource type or handling for additional fields in an existing resource type in TnT Server.
2. A new controller class if no existing controller for the type of device already exists.
3. Modification to route mapping in TnT Server to make the new interface exposed through the REST API.
4. Client-side method either in `TnTClient` or `FusionClient`, or both in the worst case.

Changes to the same modules are required each time when some driver changes its interface. Just adding a new data field in the DUT configuration settings requires new methods in several layers having nothing to do with DUT configurations. This leads to strict version dependencies and tight coupling between the layer components, which then breaks the modularity of the design.

Seemingly modular design of TnT Server has a major drawback as well: even though drivers are implemented in separate Python modules inside the package, they are still all inside the same package and code repository. This generates issues when the same component is used in multiple projects with varying targets and hardware requirements. This means that even a tiniest non-backwards-compatible change in some of the drivers means branching or forking the whole package to prevent breaking other projects. This quickly results in every project having its own flavour of the component, and those branches diverging quickly so far away from each other that merging becomes infeasible. Therefore even bugfixes often have to be implemented separately in each project, and new features can remain exclusive to a fork.

All of the hardware control is done through a single component. This does a reasonably good job in encapsulating the hardware access behind a well-defined interface, but has led to a complicated design. Also, a fault in any hardware usually means that the whole system enters into a failed state. This issue gets magnified by the design choice of initializing all configured hardware before starting the REST interface. This means that if a faulty or misconfigured hardware component fails to initialize, other applications see only that TnT Server is not able to start and have no way to query which component failed and why. Error reporting needs to be handled by either giving a generic error message to the user or tailing the log file in real-time.

These three design choices root down to the previous iteration of TnT Server which was

intended for a more singular use-case. The new version uses code from the old one, and the design is changed only partially, while the biggest effort went into migrating from Python 2 to Python 3.

4. ARCHITECTURE DESIGN OF TEST FACTORY

This chapter illustrates the new architecture design and how it aims to achieve high level of modularity. The design choices are compared to the old architecture on aspects that they differ in, and the reasons behind them are explained.

4.1 Core

Test Factory spreads out TnT Server's controller-resource -model and implements the same functionality around plugins. Each plugin implements a specific feature that corresponds to either a resource driver or a controller in TnT Server. A High-level component diagram of Test Factory is shown in figure 4.1. The figure contains only the plugins that are part of the modularity analysis of this thesis.

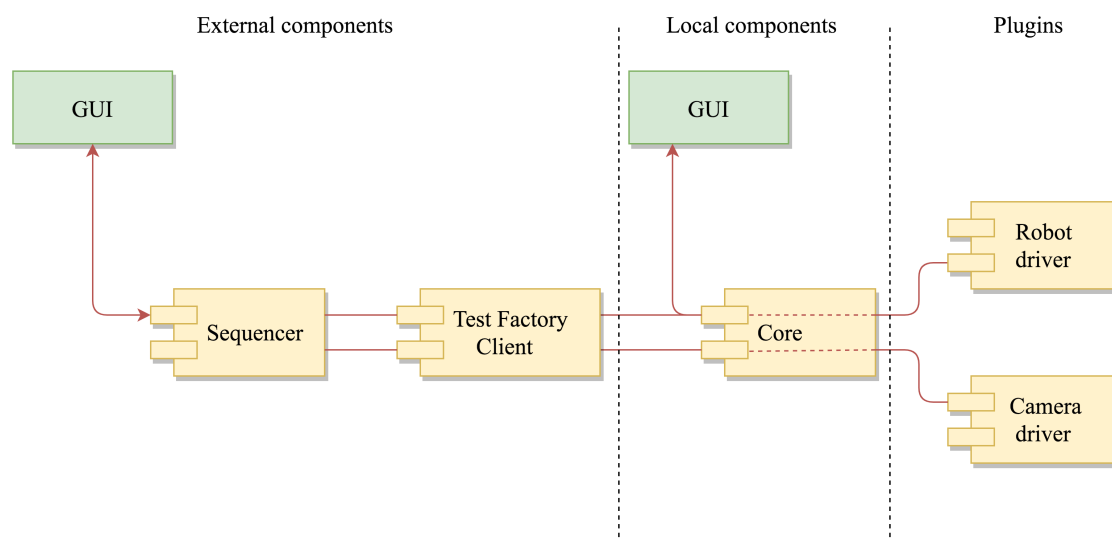


Figure 4.1 Component diagram of Test Factory software architecture

The main difference to Fusion's architecture is the extraction of the camera driver from TnT Server into its own separate module. The usage of TnT Server is reduced to coordinate calculation and robot movement, whereas DUT configuration handling is moved to the client side. In the future, robot control and coordinate system can also be split further into more self-contained units.

There are two GUI components drawn in the diagram. This illustrates how there are multiple locations to bind a user interface to. The arrow from *GUI* to *sequencer* is two-directional, whereas the arrow from *core* to the *GUI* is one-directional. This indicates that the GUI attached directly to the core can only be used in monitoring the state of the system, while the Outermost GUI can also be used to control it. The outermost GUI will also receive the same state information, as it is passed on by the client module. The intended use case for this architecture is to have the innermost GUI bound to a single Test Factory unit while the outermost GUI can control and monitor a whole cluster of them.

Test Factory is implemented in Python 3, as it is a well-understood and a widely used language within the company. As described in chapter 2, Python offers great flexibility in modular architecture design. Some of the lower-level drivers are implemented in C and C++, but all code in this project's scope is implemented in Python 3.6.

Similar to Fusion, the core of Test Factory is built around *cherrypy* and *ws4py*, of which *cherrypy* provides the HTTP-support and *ws4py* extends it to support websocket protocol. Sending and receiving of websocket messages are bound to *cherrypy*'s message bus, and the calls get routed by *DriverDispatcher* to the correct exposed method of the correct driver instance. The dispatcher handles transforming procedure call parameters from either json-rpc message or HTTP request into ones compatible with the driver function call. It also reports the errors related to not having a required driver mounted on a path, or the driver not implementing or exposing the given method. A class diagram of of Test Factory Core is shown in figure 4.2.

One important aspect of the design is that *TestFactoryCore* itself is also a subclass of *BaseDriver*. This means that it exposes its interface via the same channels as all of the drivers and that it can be thought as built-in plugin rather than a core component. It can thus utilize all the functionality given by the base class such as automatic documentation generation and instance parametrization via a configuration file.

4.2 Plugin system

Setupfile for the camera driver of Test Factory is shown in listing 4.1. Note, that the driver does not specify *test_factory_core* as a dependency in its *install_requires*-parameter. This allows installing the driver on a system that needs only the camera driver without installing the whole ecosystem.

The entry point `camera_driver.plugin:CameraDriver` defines how the class which implements the driver can be imported. In this camera driver plugin the plugin class is inside the module `plugin`, and it subclasses the *BaseDriver* class that is implemented in

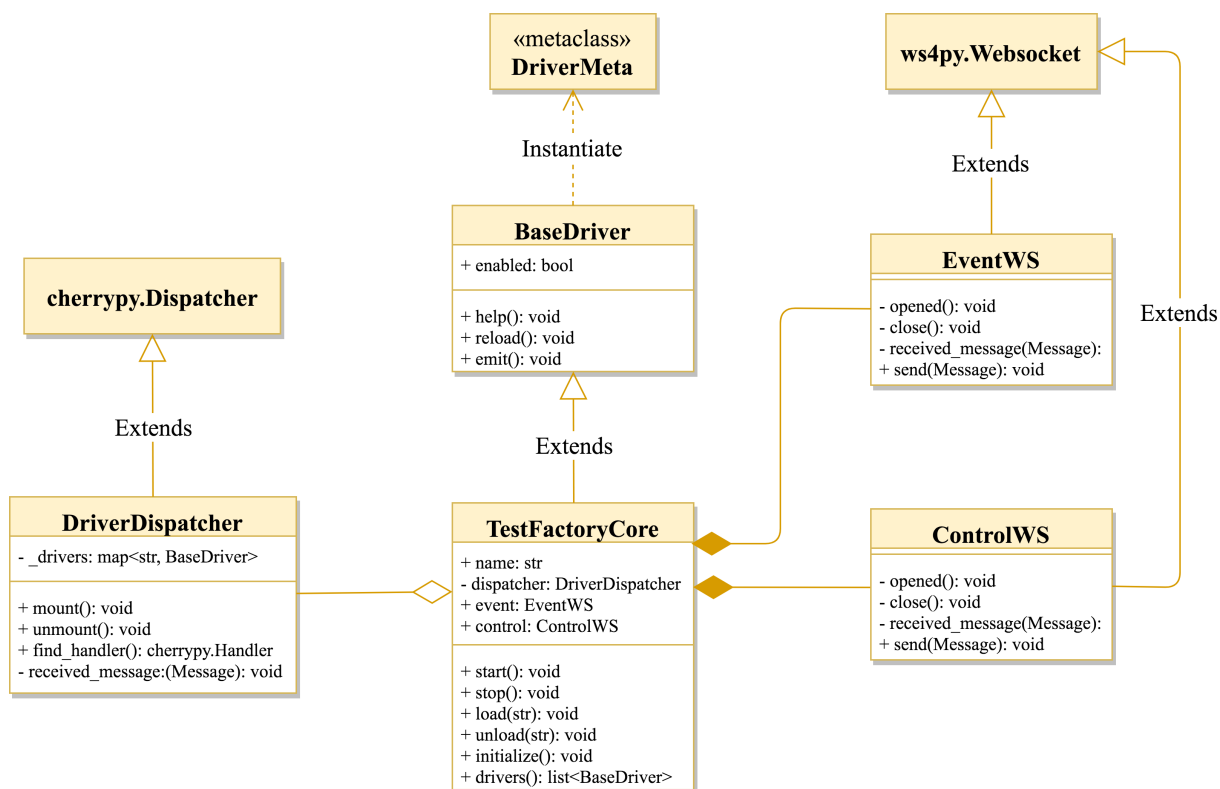


Figure 4.2 Class diagram of Test Factory Core

```

"""setup for Test Factory camera driver.
"""
from setuptools import setup, find_packages

setup(name='camera-driver',
      description='Camera driver for Test Factory',
      packages=find_package(),
      author='OptoFidelity Ltd.',
      entry_points={
          'test_factory.plugin': [
              'camera = camera_driver.plugin:CameraDriver'
          ]
      },
      install_requires=[])

```

Listing 4.1 A setup.py file of a driver plugin

Test Factory core. Low-level driver components are implemented in other modules of the camera-driver-package, and those modules do not import the plugin-module. This allows for importing of all the functionality not directly related to Test Factory without having the Test Factory core installed.

4.3 Message bus

An illustrative diagram of the message bus is shown in figure 4.3. Communication between the server and client is divided into two parts: the *control* channel and the *event* channel. The control channel is used for altering the state of the system. It supports communication via either REST calls over HTTP or json-rpc -commands over a websocket. The event channel on the other hand is only used over websockets. The channel allows for the server to have an outgoing status API that any client can connect to and follow its state.

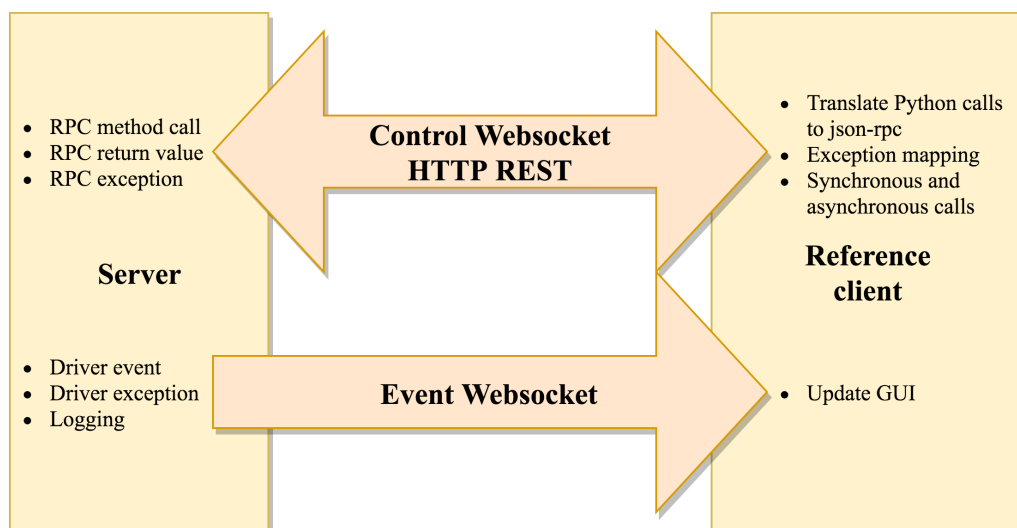


Figure 4.3 Test Factory messaging interface

The control channel is intended to be used by a single instance at a time, and the communication flows in both directions. This instance can be connected with either of the supported protocols. The server does not enforce the limit of one client per session, so the responsibility is on the programmer scripting the client side to not cause race conditions or a conflicting state.

Exception to the previous principle are requests that do not modify the state of the server. These include state queries and data retrievals. When using the REST interface these methods are implemented as GET-methods and can be clearly recognized, but using the websocket interface the “const”-methods are defined only in the documentation. These non-state-modifying API endpoints form a third channel, *data* channel, that shares the

road with control channel. Different message types supported in all three channels are listed in table 4.1.

Table 4.1 Websocket message types

Message type	Channel	Purpose
Method call	control/data	Call a method in driver API or fetch a resource from a driver.
Response	control/data	Return from a method in driver API or return the requested resource.
Event	event	A general event in a driver. Follows the standard of json-rpc 2.0
Exception	control/event	An error has occurred when executing a command. It is sent as an event to all listeners and as a return value to caller in control channel. Follows the standard of json-rpc 2.0.

The event channel is a one-way communication interface that can server multiple clients simultaneously. Unlike the control and the data channels, the communication is initiated from server side and thus the event channel cannot be used through the REST API. Drivers can use it to publish any messages, such as log entries or error reports. Driver can also announce that it has a new resource available through the data channel.

Messages published on the event channel are asynchronous. They get pushed to the cherry-py's message bus and cherry-py will send the message when it decides to do so. This also means that the driver can keep up executing at full speed and does not have to worry about the time it takes to serialize the message and send it through the socket. Another driver can hook a callback method onto a specific message type and this callback will not affect the execution of the driver that was hooked into.

Typical use-case for the event channel is to have a graphical user interface that shows the user what is currently happening in the system. It could for example display all the images captured with the camera immediately without disrupting the normal test sequence. It is also trivial to implement a log view in the GUI to see the state of the system in more detail.

4.4 Driver interface

Drivers are encouraged to implement a few standard methods. These include commands to initialize the device, close the device and query the state of the device. By having all drivers implement the same basic methods the system can execute these commands for all bound drivers at once, in parallel. This is a significant improvement over the previous system where all of the hardware drivers were initiated sequentially and the API was

opened only after all drivers were loaded successfully. With Test Factory, an issue with a device driver can be detected and identified in client side as the failure in the driver's initialize-method occurs while the client is already listening for error events.

Sequence diagram 4.4 shows the startup steps of the API. The execution starts from Python process loading the main module of `test_factory_core`-module. It then creates an instance of `TestFactoryCore` which then becomes the first driver instance that exists in the process. The driver cannot be used yet, though, since there is no dispatcher active, and `cherryPy`'s bus is not running. `TestFactoryCore` creates the `DriverDispatcher` on its own and mounts it to the root of `cherryPy`'s tree. After it is complete, it calls the `block`-method of `cherryPy`'s engine and thus transfer control to `cherryPy`.

From this point onwards the core API is open for clients to connect. If the process is running on Linux, a message is written into `systemd`'s notification socket to indicate that other units dependent on this service being up can now be launched. Everything until this moment should be as fail-proof as possible: no hardware components are initialized, and no configuration files are loaded yet. At this point, a client can connect to the event channel and get information of the system's state.

After the API has been started the server begins to wait for a client to connect to its control channel. The first task of a connected client is to initialize the hardware components and load configuration files. It is upto the client to select which components get initialized and what to do in case of an error. In the example setup of figure 4.4 the `initialize`-method of `TestFactoryCore`-driver is called by the client which triggers initialization of all configured devices.

Figures 4.5 and 4.6 demonstrate image capturing on both the old and the new architectures. In these sequence diagrams, it is easy to see how Test Factory simplifies the interaction between the client and the driver.

The sequence grabs a burst of three camera frames. The target is to capture three images with identical settings as quickly as possible. With Fusion there are two issues immediately visible: there are five different entities that need to know that there exists a method for taking a still image. and the whole stack needs to be traversed back and forth entirely for every single frame. Image capture is split into two parts: grabbing a frame and then fetching the recently grabbed frame from the driver. These two do not have to be done in pairs; one can grab multiple frames and then fetch them after all of them are fetched.

Test Factory fixes the previously mentioned issues by flattening the server side and making the client side more dynamic. There is only one instance in the server side that needs to know that there is a `take_still`-method available. The client side can use Python's

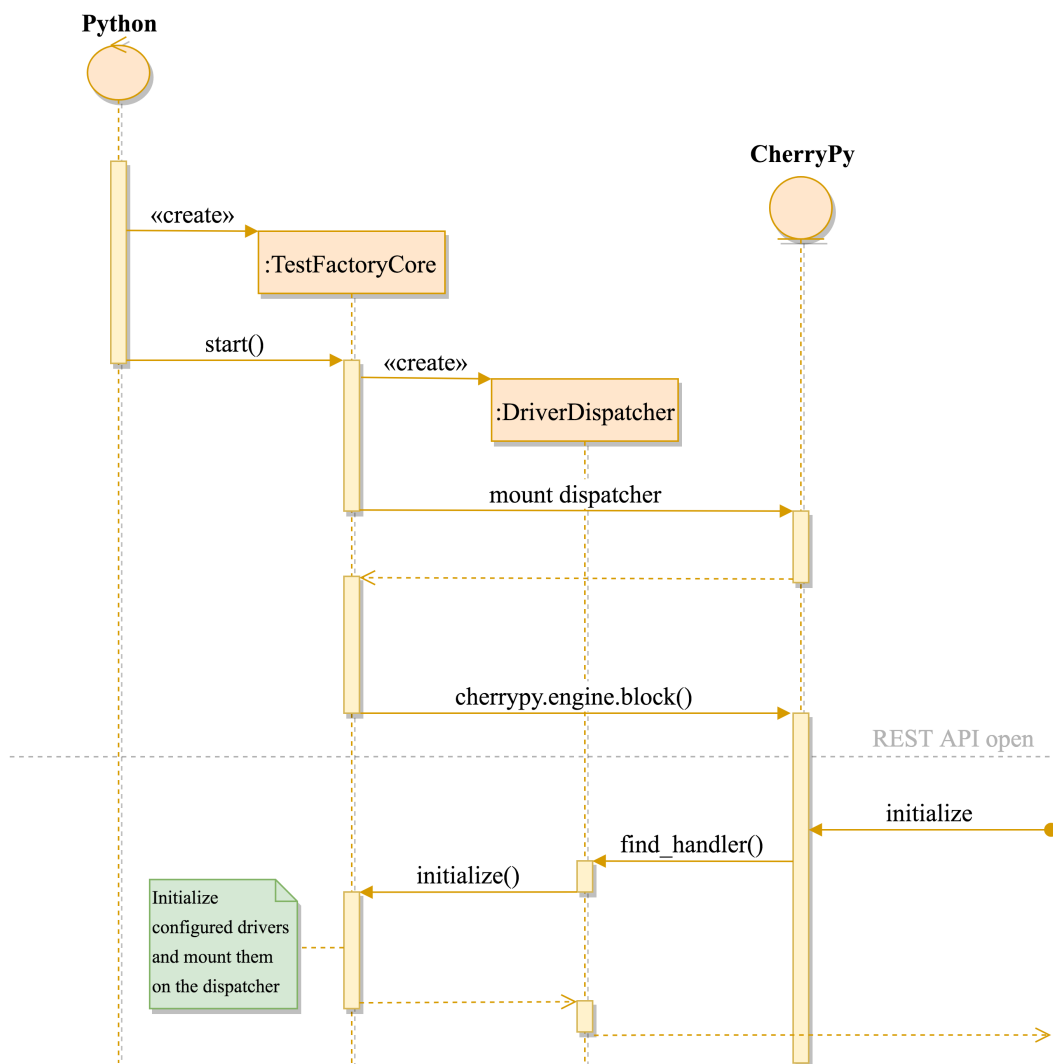


Figure 4.4 Sequence diagram of the startup of Test Factory

dynamic attribute access to allow it to not know about the method names provided by the drivers mounted on the server. This results in only two locations being dependent on the name of the method: the driver itself and the code implementing business logic by using the client.

4.5 Configuration

The main configuration for Test Factory is done in `drivers.conf`-file. This file defines what devices will be available to the system, how they are initialized and where they are mounted in the API tree. An example of this configuration file is shown in listing 4.2.

The configuration file resides locally on the same machine as the API is running on. In a system of multiple Test Factory units, each one of them has its own driver configuration file. The configuration file can thus contain calibration values that may not be equal

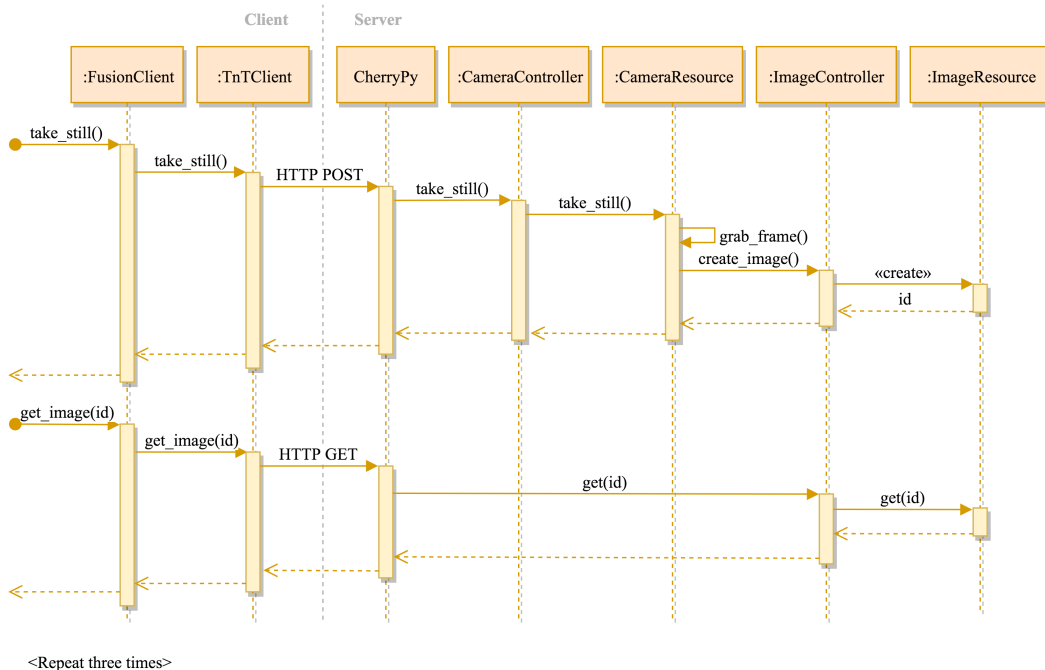


Figure 4.5 Capturing three images on Fusion

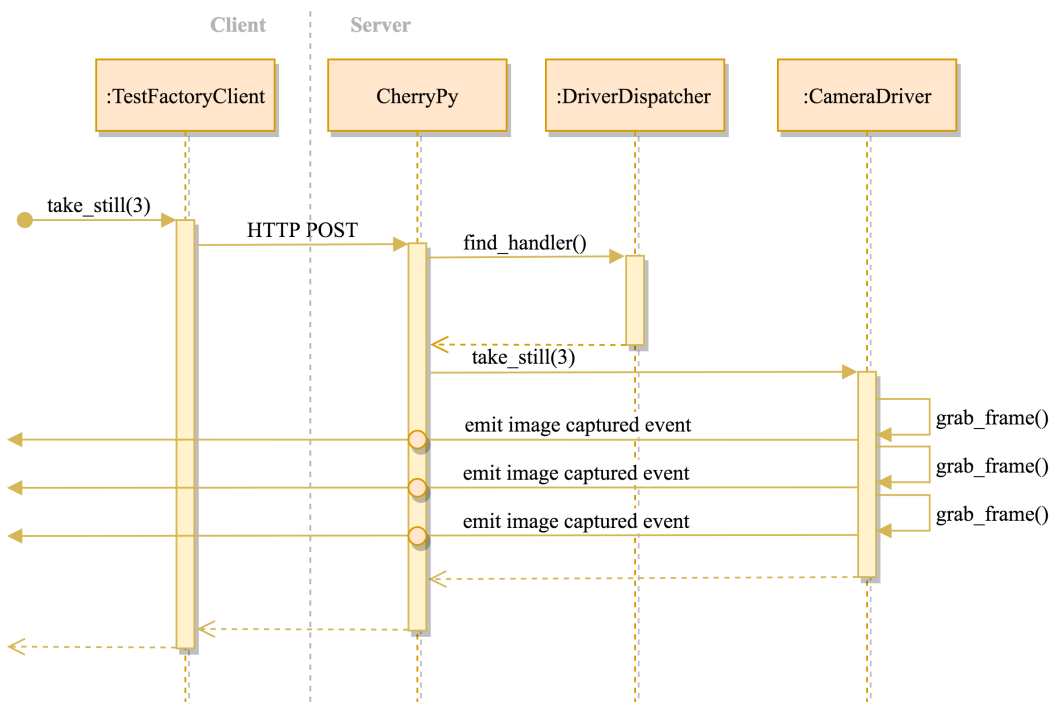


Figure 4.6 Capturing three images on Test Factory

```
[main-camera]
Device = camera
FocalLength = 8
FocusDistance = 50.2
IpAddress = 10.10.0.2
PacketSize = 8192

[secondary-camera]
Device = camera
FocalLength = 4
FocusDistance = 30.15
IpAddress = 10.10.0.6
PacketSize = 8192
```

Listing 4.2 Sample configuration file

between the robots. In the example configuration file, one of these kind of variables is the *FocusDistance*. The camera used in this project does not have a focusing system that could be controlled programmatically, but instead it has to be adjusted manually with the focus in on the lens. This will result in a slightly different real focus distance in each camera, which then can be compensated with robot movement while we know the exact focus distances.

It is also possible to define multiple devices that use the same driver with different parameters, and that is exactly what has been done in the example configuration. The configured system has two cameras: the main camera and the secondary camera. These cameras are different from hardware perspective, they have for example a different focal length.

In a system of multiple Test Factory units the configuration system can be centralized into a common server which shares the configuration files to multiple machines. In that system, common settings are stored in the network location, while individual units can define the same attributes in their local configuration files overriding the global options. This allows storing both settings-related (values usually do not differ between units) and calibration-related (values usually differ between units) parameters in the same file.

4.6 Logging

The new architecture design does not come without issues. When scattering the pieces around different components — possibly even different processes — it becomes increasingly difficult to implement features that require centralized control. One example of this is logging. Logging from multiple processes easily creates multiple separate log files laying around in the file system, and following the state of the whole system is hard.

Test Factory tries to solve this issue with several methods. Firstly, logging is tied to the message bus architecture. Log entries are sent as events from drivers regardless of which component or process generates them. This way the log entries can be collected in a single place and written into a single log file. The process reading the logger events can also run on a different machine in the network. The second method to prevent scattered log entries is to use system journal if it is available. Since the target operating system for Test Factory is Linux and the intended way of running the server is via a *systemd* service, all standard output of the process will get written into system log locally.

5. EVALUATION

This chapter takes the two systems introduced in the previous chapters, and performs the modularity analysis defined in chapter 2 on them. Since increasing the modularity was one of the main drives for the new architecture, this gives a fairly good first impression on the successfulness of the redesign.

The complete systems containing all components that form Fusion and Test Factory have a massive amount of modules and classes. That is why the modularity analysis is conducted on a subset of modules that contribute to a system similar to the example system shown in figure 1.1, which makes the scale of the research more suitable for a master's thesis, and makes it more feasible to perform the analysis without automated tools. The system we are inspecting consists of the Test Factory core, camera driver and a stub representing the robot driver. Robot control has been abstracted away by defining it only as `RobotController` in Fusion, and `RobotDriver` in Test Factory context. In reality, both systems have a complex set of modules behind that functionality. Test Factory can even use `TnTClient` as the backend for robot control. From Fusion, we selected the modules that contribute to the same functionality.

5.1 Cohesion

As defined in chapter 2.2.1, to calculate cohesion of the software architecture, we need to measure the individual cohesion for every single method in every single class in the part of the system which is under the analysis. Class cohesions calculated with formulas 2.1 and 2.2 are listed in table 5.1 for Fusion and 5.2 for Test Factory.

From the values listed in the forementioned tables the total cohesion of both systems can be calculated. For Fusion it yields a score of 0.320 and for Test Factory a score of 0.598. So Test Factory scores a significantly higher value.

Cohesion of Fusion	0.320
Cohesion of Test Factory	0.598

Table 5.1 Cohesion of Fusion

Class	Cohesion
FusionClient	0.394
FusionRobotClient	0.055
FusionCameraClient	-
TnTClient	1.000
TnTRobotClient	-
TnTCameraClient	0.167
RobotController	-
CameraController	-
Camera	0.237
Robot	0.066

Table 5.2 Cohesion of Test Factory

Class	Cohesion
TestFactoryClient	0.500
DriverDispatcher	1.000
BaseDriver	0.278
TestFactoryCore	1.000
EventWS	-
ControlWS	-
CameraDriver	0.367
RobotDriver	0.444

There are four modules in the tables that do not have a value. This is because the classes have no attributes, so they do not contribute to the cohesion. In Fusion's context these are the controller classes inside TnT Server. Those classes are implemented as cherrypy controllers and they access all of the required objects through global functions instead of class attributes. Measuring cohesion of a such class is not feasible with the selected method. Instead, for each method, the signatures of the calls to those global functions would have to be analyzed.

Fusion has classes that have a very low cohesion value (< 0.2). That is caused by the unnecessary layers in the client-server stack of Fusion. As discussed in chapter 3.3.2, there are several places that require changes when a new functionality is added; this is clearly visible in the cohesion values as these classes just pass functionality to the lower layers and do not have any logic in themselves. Eventhough they contain a big number of member variables, most of them are used in a single function, which uses that attribute and nothing else.

Table 5.5 Coupling values of Test Factory

	<i>test_factory.client</i>	<i>test_factory.core</i>	<i>test_factory.driver</i>	<i>test_factory.api</i>	<i>camera_driver</i>	<i>camera_driver.test_factory_driver</i>
<i>test_factory.client</i>	-	0	0	0	0	0
<i>test_factory.core</i>		-	2	2	0	0
<i>test_factory.driver</i>			-	2	0	2
<i>test_factory.api</i>				-	0	2
<i>camera_driver</i>					-	3
<i>camera_driver.test_factory_driver</i>						-

Table 5.6 Number of interconnections in Test Factory

	<i>test_factory.client</i>	<i>test_factory.core</i>	<i>test_factory.driver</i>	<i>test_factory.api</i>	<i>camera_driver</i>	<i>camera_driver.test_factory_driver</i>
<i>test_factory.client</i>	-	0	0	0	0	0
<i>test_factory.core</i>		-	3	14	0	0
<i>test_factory.driver</i>			-	4	0	3
<i>test_factory.api</i>				-	0	3
<i>camera_driver</i>					-	12
<i>camera_driver.test_factory_driver</i>						-

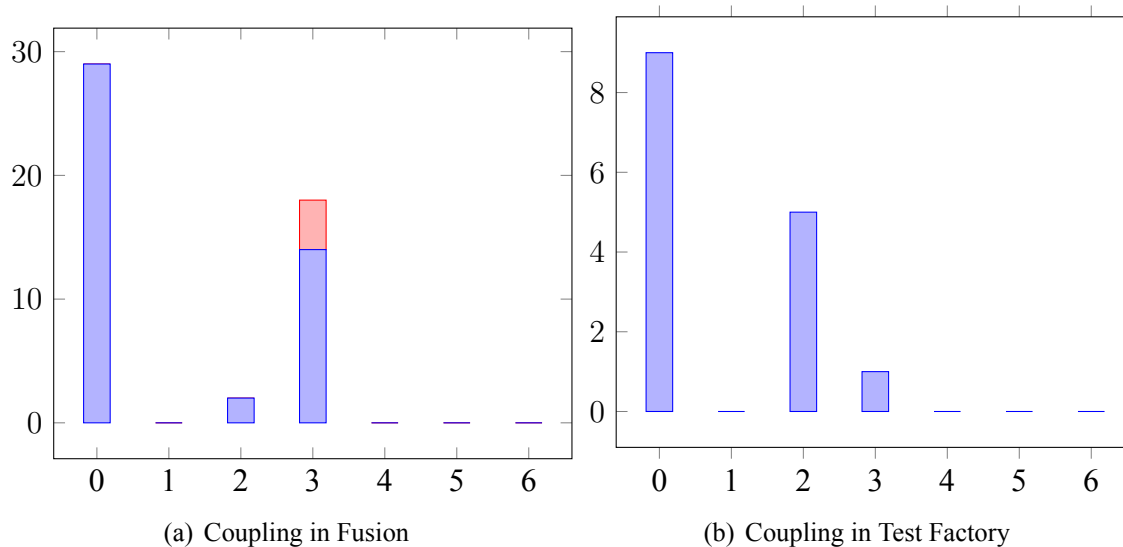


Figure 5.1 Histograms describing the distribution of coupling values of the target systems. The blue bar represents direct coupling, and the red bar indirect coupling.

By applying the formula 2.6 we can get the total coupling of 0.527 for Fusion and 0.347 for Test Factory. The value is significantly lower for Test Factory, largely due to the client having no direct nor indirect coupling to any of the modules in the server side.

Coupling of Fusion	0.527
Coupling of Test Factory	0.347

The histograms in figure 5.1 show how the new architecture reduces the coupling in the system. The histogram 5.1 (a) visualizes the coupling levels in Fusion, as the histogram 5.1 (b) does the same for Test Factory. These numbers are “raw”, taken before formula 2.5 has been applied. When the numbers are considered as proportional to the number of modules, it is clear that the amount of modules with a coupling level three has decreased, especially when indirect coupling is taken into account.

One clear difference that is not directly visible in the numbers but is easy to spot from the tables, is that the Test Factory achieves the same functionality with a way fewer amount of modules. A smaller number of modules does not directly guarantee a more modular system but it enhances the clarity and maintainability of it. The effect of indirect coupling is also visible: even though two modules do not directly interact with each other, they are part of a long method chain chain, all layers of which have to follow the same parameter format. This means that the top-level client-side method in `fusion_camera_client` has to know what is the parameter structure of the lowest level method in `resources.camera`,

as do all the intermediate steps in between. The effect of indirect coupling in this research was not as big as expected, though. Test Factory does not have any indirect coupling, as its client uses dynamic attribute access and variable arguments to generate API calls, and its class hierarchy is more flat.

5.3 Reconfigurability

The reconfigurability of the systems was not measured numerically. Instead, the reconfiguration capabilities are judged by discussing how easy it would be to deploy the same software with a completely different set of drivers.

TnT Server implements good configurability in itself, but the upper layers (`fusion_client` in particular) expects a specific driver and hardware configuration to be present. This renders the reconfigurability of Fusion to essentially zero. Theoretically, it is possible to leave out some hardware components while the drivers are still bundled in the system, but that would not be reconfigurability from the software's point of view.

Test Factory, on the other hand is reconfigurable by design. The core does not know anything about any of the drivers, so it cannot expect anything to be present.

6. CONCLUSION

This research aimed to answer the question: *How to increase modularity and configurability of a robotics system?* It took two software platforms that tried to solve a similar problem and compared them by their *cohesion* and *coupling*. The results are summarized in the table 6.1.

Table 6.1 Summary of the research.

	Fusion	Test Factory
Cohesion	0.320	0.598
Coupling	0.527	0.347
Reconfigurability	no	yes

The new architecture was shown to increase the modularity of the system, as the amount of cohesion increased significantly and the amount of coupling decreased a fair amount. An equally large effect comes from the simplification the new design provided, though: there are now much fewer modules needed to perform the same functionality and more. These results conclude that the choice of plugin architecture was successful.

Industrial robotics and test automation were the context of this research, and their effects on the analysis were deliberated. It was shown that the closeness to the hardware and challenging environment of a factory line makes for a different basis to create a functional software architecture. Error handling becomes more critical when there is a major chance of an expensive measurement device or DUT being destroyed because of a software malfunction.

Another target of the research was to study about the feasibility of doing the modularity analysis to a dynamic language when there exists no compiler to automatically generate data about the codebase. It was seen to be a lot of manual work to calculate cohesion and coupling of modules by hand. It is sometimes hard to define how two modules are linked to each other. With duck-typing it is possible for a module to depend on the internals of another module, but still to be able to fallback to some other behavior when the desired module does not have the expected internals.

6.1 Validity of the study

There are several aspects of the research that reduce its validity. These affect the repeatability and correctness of the measurements.

Firstly, instead of relying to automated tools most measurements are done by hand and for example defining the level of coupling between two modules is purely based on the judgement of the author. Someone else performing the research on the very same codebase might get a slightly different results.

Secondly, Test Factory as a platform is still a work in progress and no full system has yet been manufactured or tested. This means that the software architecture, eventhough it is been designed fairly extensively, is still subject to change. Even minor modifications to the architecture can massively impact the calculations performed in the previous chapter. These changes may originate either from an internal or external need; a pilot customer may demand a different approach, or the requirements may change from the company side. Additionally, a new proof of concept does not have all the workarounds and hacks that a mature system inevitably houses. Only time will tell to which direction the new architecture will go, but at least the starting point is better than the original.

The repeatability of the research can be questioned because all of the source code in the project is proprietary. The codebase is owned by OptoFidelity and is not available to the public. This means that to perform the exact same research one would have to first deal with legal issues. The author of this paper is an employee of OptoFidelity and thus has access to all of the source code.

6.2 Further research

There is a lot of room for further research on the subject of measuring modularity of software platform implemented with a dynamic programming language.

Tools have been designed for Python that analyse an existing code base semantically, one example being Jedi. Jedi is a library that is designed to be used as a utility plugin for programming editors providing auto-completion, definition-lookups, linting and basic refactoring. It supports majority of the most used text editors and IDEs available. Its support for static code analysis could easily be leveraged in this kind of research work, as it is able to build a semantic understanding of the codebase. Extending the tool to output information on the modularity of the components would certainly be possible.

Python has recently introduced features that are influenced by statically typed languages. There is now a decent support for type annotations and while they are not enforced by the

language, they are a good hints for the programmer and makes it easier to use a library by looking at the function declarations. As the annotations are available programmatically, tools such as Jedi can also utilize them to provide more accurate data. Mypy-module takes this a step further and provides a “compilation step” where it does type checking before delivering the code.

These tools may not have a large effect on the results of the research but they would definitely have made the modularity analysis easier. It would have at least ruled out the human error in the measurements, and in the best case, made the analysis completely automatic. In that case the research would be trivial to implement on a different product.

REFERENCES

- [1] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003.
- [2] A. Bolour, “Participants of a plug-in extension.” 2003. [Online]. Available: https://www.eclipse.org/articles/Article-Plug-in-architecture/images/eclipse_extensions.jpg
- [3] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 06 1994, copyright - Copyright Institute of Electrical and Electronics Engineers, Inc. (IEEE) Jun 1994; Last updated - 2011-07-20; CODEN - IESEDJ. [Online]. Available: <https://search-proquest-com.libproxy.tut.fi/docview/195574529?accountid=27303>
- [4] B. J. Cox, “Planning the software industrial revolution,” *IEEE software*, vol. 7, no. 6, pp. 25–33, 1990.
- [5] David Halter, *Jedi - an awesome autocompletion/static analysis library for Python*, 2018. [Online]. Available: <https://jedi.readthedocs.io/en/latest/>
- [6] R. Dörbecker, D. Böhm, and T. Böhm, “Measuring modularity and related effects for services, products, networks, and software – a comparative literature review and a research agenda for service modularity,” in *2015 48th Hawaii International Conference on System Sciences*, Jan 2015, pp. 1360–1369. [Online]. Available: <https://doi.org/10.1109/HICSS.2015.167>
- [7] N. Fenton and A. Melton, “Deriving structurally based software measures,” *Journal of Systems and Software*, vol. 12, no. 3, pp. 177–187, 1990.
- [8] G. Van Rossum, B. Warsaw, and N. Coghlan, *Style Guide for Python Code*, Python Software Foundation, 2001. [Online]. Available: <https://www.python.org/dev/peps/pep-0008/>
- [9] J. G. Garcia, J. G. Ortega, A. S. Garcia, and S. S. Martinez, “Robotic software architecture for multisensor fusion system,” *IEEE Transactions on Industrial Electronics*, vol. 56, no. 3, pp. 766–777, March 2009. [Online]. Available: <https://doi.org/10.1109/TIE.2008.2007014>
- [10] J.-F. Gélinas, M. Badri, and L. Badri, “A cohesion measure for aspects.” *Journal of object technology*, vol. 5, no. 7, pp. 75–95, 2006.

- [11] Google Android, “Photo of the nexus 5,” 2013. [Online]. Available: https://commons.wikimedia.org/wiki/File:Nexus_5_Front_View.png
- [12] G. Gui and P. D. Scott, “New coupling and cohesion metrics for evaluation of software component reusability,” in *2008 The 9th International Conference for Young Computer Scientists*, Nov 2008, pp. 1181–1186. [Online]. Available: <https://doi.org/10.1109/ICYCS.2008.270>
- [13] M. J. Harrold and P. Kolte, “A software metric system for module coupling,” *J Syst Softw*, vol. 20, no. 3, pp. 295–308, 2003.
- [14] M. Hitz and B. Montazeri, “Measuring coupling and cohesion in object-oriented systems,” 1995.
- [15] A. Kleiner and T. Buchheim, “A plugin-based architecture for simulation in the f2000 league,” in *RoboCup 2003: Robot Soccer World Cup VII*, D. Polani, B. Browning, A. Bonarini, and K. Yoshida, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 434–445.
- [16] S. Laine, H. Kuosmanen, and K. Jokinen, “Calibration device for camera – finger -offset,” U.S. Patent 20 180 089 522, March, 2018. [Online]. Available: <http://www.freepatentsonline.com/y2018/0089522.html>
- [17] U. Larsson, J. Forsberg, and A. Wernersson, “Mobile robot localization: Integrating measurements from a time-of-flight laser,” *IEEE Transactions on Industrial Electronics*, vol. 43, no. 3, pp. 422–431, 1996.
- [18] J. Liebig, S. Apel, A. Janke, F. Garbe, and S. Oster, “Handling static configurability in refactoring engines,” *Computer*, vol. 50, no. 7, pp. 44–53, 2017. [Online]. Available: <https://doi.org/10.1109/MC.2017.212>
- [19] A. Lucke, “Svs-vistek - svcam-eco blackline - tubus,” 2011. [Online]. Available: https://commons.wikimedia.org/wiki/File:SVCam-ECO_Series_black_with_Tubus.JPG
- [20] M. D. McIlroy, J. Buxton, P. Naur, and B. Randell, “Mass-produced software components,” in *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, 1968, pp. 88–98.
- [21] G. J. Meyers, “Reliable software through composite design: Petrocelli,” *Charter, New York*, 1975.
- [22] Mozilla, “Firefox 57.0 release notes,” 2017. [Online]. Available: <https://www.mozilla.org/en-US/firefox/57.0/releasesnotes/>

- [23] K. Needham, “The future of developing firefox add-ons,” 2015. [Online]. Available: <https://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/>
- [24] OptoFidelity Ltd., “OptoFidelity™ Fusion Tester,” 2018. [Online]. Available: https://www.optofidelity.com/wp-content/uploads/2018/03/Optofidelity-FusionBlue_46-e1521674921191.png
- [25] —, “OptoFidelity™ Test Factory,” 2018. [Online]. Available: <https://www.optofidelity.com/wp-content/uploads/2018/02/testfactory2.jpg>
- [26] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical Software Engineering*, vol. 14, no. 2, p. 131, Dec 2008. [Online]. Available: <https://doi.org/10.1007/s10664-008-9102-8>
- [27] M. Rüßmann, M. Lorenz, P. Gerbert, M. Waldner, J. Justus, P. Engel, and M. Harnisch, “Industry 4.0: the future of productivity and growth in manufacturing industries,” *Boston Consulting Group*, vol. 9, 2015.
- [28] Z. Salcic, U. D. Atmojo, H. J. Park, A. T. Y. Chen, and K. I.-K. Wang, “Designing dynamic and collaborative automation and robotics software systems,” *IEEE Transactions on Industrial Informatics*, pp. 1–1, 2017. [Online]. Available: <https://doi.org/10.1109/TII.2017.2786280>
- [29] E. Sandewall, “The pipelining transformation on plans for manufacturing cells with robots.” in *IJCAI*, 1987, pp. 1055–1062.
- [30] P. Tonella, “Concept analysis for module restructuring,” *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 351–363, Apr 2001. [Online]. Available: <https://doi.org/10.1109/32.917524>
- [31] J. Villalobos, “The road to firefox 57 – compatibility milestones,” 2017. [Online]. Available: <https://blog.mozilla.org/addons/2017/02/16/the-road-to-firefox-57-compatibility-milestones/>
- [32] S. Wagner, S. Winkler, E. Pitzer, G. Kronberger, A. Beham, R. Braune, and M. Afenzeller, “Benefits of plugin-based heuristic optimization software systems,” in *Computer Aided Systems Theory – EUROCAST 2007*, R. Moreno Díaz, F. Pichler, and A. Quesada Arencibia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 747–754.
- [33] J. Wan, S. Tang, Z. Shu, D. Li, S. Wang, M. Imran, and A. V. Vasilakos, “Software-defined industrial internet of things in the context of industry 4.0,” *IEEE*

Sensors Journal, vol. 16, no. 20, pp. 7373–7380, Oct 2016. [Online]. Available: <https://doi.org/10.1109/JSEN.2016.2565621>

- [34] F. Xia, L. T. Yang, L. Wang, and A. Vinel, “Internet of things,” *International Journal of Communication Systems*, vol. 25, no. 9, pp. 1101–1102, 2012.