TDS: TRADR Display System

A modular interface for robot-human team coordination during urban search and rescue operations

by

J. Sparreboom
M.C. de Graaf

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on Thursday September 3, 2015 at 01:30 PM.

Thesis committee:	Coach:	Dr. J. de Greeff,	TU Delft
	Client:	Dhr. I. Stel,	TNO
	Coordinator:	Dr. Ir. F.F.J. Hermans,	TU Delft

Foreword

This document serves as final deliverable of the development of the TRADR Display System. This development was done for the TI-3806 course of the Bachelor Computer Science program at the Delft University of Technology in the Netherlands. This course, better known as "Bachelor's Thesis" or "BEP", concludes the Bachelor's program by letting groups work on a real program for a real client. The goal of the course is for the partaking students to gain experience from a realistic working environment, from start to end.

This project was issued by J. de Greeff, from the TRADR consortium. It began at first of June 2015 and came to an end at September 3nd 2015. The team working on this BEP consists of two persons: Marnix de Graaf and Jamey Sparreboom. The research and development mainly took place at the faculty of Electrical Engineering, Mathematics and Computer Science at the Delft University of Technology. Work has also been done for a brief period at the Fraunhofer Institute at Sankt Augustin and TNO Soesterberg.

The contents of this report consist of an introduction of the faced problem and a description of the used methodology for solving the problem. It also features an elaboration of the three phases the project was split in; the research-, development and concluding phase.

The TU Delft coach for this project is J. de Greeff. The client is the TRADR consortium. Mr. I. Stel has been the contact person for the TRADR consortium. All arrangements with J. de Greeff, also part of the TRADR consortium, can also be seen as contact with the client. The Bachelor Coordinator assigned is F.F.J. Hermans.

We would like to thank the following persons:

- J. de Greeff, for guiding us through sometimes difficult times and helping us in every way needed with a smile.
- I. Stel, for giving valuable feedback on our work and allowing us to get a view on a company like experience.
- F.F.J. Hermans, for answering our project and organization related questions quickly.
- R. van Hattem, for being our savior in technical emergencies.
- D. Reuter, for kickstarting the project to have a working base and fun times at Fraunhofer.
- E. Zimmerman, for giving us a clear view on the problem and the broader picture.
- T. Bagosi, for giving extensive help with the database.

Marnix de Graaf
Jamey Sparreboom
Delft, August 2015

Contents

1 Summary

This Bachelor Thesis took place as part of the Bachelor Computer Science at Delft University of Technology. This report describes a project issued by the Long-Term Human-Robot Teaming for Robot Assisted Disaster Response (TRADR) consortium. During this project, the team worked on creating a user interface for use in human-robot teams during urban search and rescue missions and situation assessment.

New innovations for use in search and rescue operations are being created to keep rescue workers safe during these operations. It also allows them to work more efficient, possibly saving more lives. One of these innovations is the use of robots to assess the situation at disaster areas. To work efficiently in robot-human teams, a clear overview of the data gathered should be available to the rescue workers. This project is aimed at creating a graphical user interface to give the rescue workers this overview.

The TRADR Display System (TDS) is a user interface created using Python, RQT and QT. It consists of an overhead map of the disaster location using the virtual globe application Marble. This overhead map shows gathered and shared information in an efficient manner using a Point Of Interests (POIs) pinned on the map. Beside the overview map the user interface consists of several tools and informational toolbars to allow further elaboration on the shown points of interest.

The development of the TDS was done in three phases. The first phase is defined as the research phase in which the system was designed and research has been done to make sure the provided solution is the best solution. Here it has been found that the user interface should be not distracting which is solved by the option to hide information and giving non-invasive messages.

The second phase is defined as the development phase, where the program was created using the insights gathered from the research phase. The application is build up as set of plugins for modularity. The application, which is a plugin itself, has been built in a model-view-presenter architectural pattern. It has been made easy to add a new plugin for new developers in the project.

The last phase is the concluding phase, in which the product created in the development phase was tested and documentation was created. All 'must have' requirements have been implemented. The code was tested by Software Improvement Group (SIG), a company which performs dynamic code analysis. The result is that the code is above average maintainable (a four out of five stars on SIG's star system), but there are not enough tests present.

The result is a modular graphical user interface for use in urban search and rescue missions, which allows rescue workers to keep a clear overview, allowing them to work more efficiently in robot-human teams.

2 INTRODUCTION

Search and Rescue missions are missions aimed at searching for and providing aid to persons in direct danger. During these missions one or multiple groups of rescue workers go to the dangerous areas in a coordinated fashion.

Search and Rescue can be split in multiple subgroups. These subgroups include Mountain Search and Rescue, Ground Search and Rescue and Urban Search and Rescue. Urban Search and Rescue (USAR) is the type that aims at saving persons in need of rescue at locations such as collapsed buildings or dangerous industrial sites. These situations often occur due to earthquakes or extreme weather conditions, but explosions are also a possible cause.

USAR missions often take place in dangerous environments, therefore the safety of the rescue workers is an important aspect. To keep the rescue workers as safe as possible, they are well trained for these specific environments. This training allows them to assess locations quickly and notice dangers like unstable debris or fires. Assessing the location is important to get an insight of the hazards during the rescue. This assessment is done by taking samples of for example leaking unknown substances and spotting multiple possible hazards by sensing, such as seeing a broken power cable. But to assess these locations the rescuers workers are still required to enter the dangerous area.

Many new innovations, like tools for the rescue workers, have been created to make the Search and Rescue operations more efficient and safer for these rescue workers. Some of these innovations include the use of Unmanned aerial vehicles (UAVs) and Unmanned ground vehicles (UGVs) in the dangerous areas to assess. These machines are able to show many different kinds of data to the Search and Rescue team in the form of video, audio, thermal data, etc. With access to this data it is possible to provide the team with knowledge about the location of hazards before entering the mission area. Another advantage is that the robots can get or fly to places inaccessible for humans.

In a USAR team, there often are multiple roles divided through a hierarchy, specifying who leads a certain subgroup of rescuers. These roles are based on the specialties of every individual team member. A USAR team in the scope of this project can contain for example:

- a Team leader who organizes the teams and plans their actions
- a UAV/UGV operator who controls the robot or searches for victims or hazards using the data received from the robot
- an in-field rescuer responsible for assessing the situation by entering the dangerous area, aiding victims and many other possible tasks according to his or her specialty.

All these team members need different information during their work. The UAV/UGV operator will need to receive all the robot's data, whereas an in-field rescuer requires to know what there has been discovered about the situation. The team leader needs the oversight of all this data.

The Long-Term Human-Robot Teaming for Robot Assisted Disaster Response (TRADR) consortium is working on software to allow easier and safer situation assessment using robot-human teams.[1] To keep a clear overview of the current situation and every actor during a mission, it is important that available information is transmitted to all search and rescue (S&R) team members who require it. In most of the cases, this is the whole team. This information includes photos made by UAVs and UGVs and detailed descriptions given by in-field rescuers who took a better look at a location. The S&R team has to work as fast as possible and therefore every moment spent on searching for information can cost lives. Quick access to information shared by other team members is crucial.

To do this, a modular Graphical User Interface (GUI) for the TRADR system is desired, on which a map of the area can be displayed, along with various other information such as locations of casualties, danger zones, photos of disaster sites etc. A modular GUI is a user interface in which all components, such as toolbars, layers, information views or locations can be turned on and off independently. This information should be easily accessible by the in-field rescuers and new information should be easy to insert and send to all other emergency responders involved in the mission.

This GUI for the TRADR-project is called the TRADR Display System (TDS) and consists of two major parts; the MapView and the Operator Control Unit (OCU). The OCU is already created by another partner in the TRADR project and allows users to control a robot using a GUI. This bachelor project is aimed at creating the MapView plugin to run alongside the OCU. This plugin is required to be extendable, therefore it will need a modular design. It also will need to be a sturdy and easy to use system, which requires no deep technical knowledge. The main question of the project therefore is:

How to create a modular GUI for use in a robot-human-team-based Search and Rescue operation?

To answer this main question other (sub)questions have to be asked.

1. How to create a modular GUI?
2. What is an effective way to display the information needed by these rescuers?
3. How to integrate this system into existing systems TRADR already uses?

To address these issues, firstly the used methodology for this project is discussed in chapter 3. Then the results of the beginning of the project, the research phase, are covered in chapter 4. After the research phase followed the actual development of the GUI and all of its parts which is examined in chapter 5. The project concludes with the concluding phase as can be found in chapter 6. The answer to the research question is elaborated upon in the conclusion in chapter 7 and a discussion about the project is discussed in chapter 8. For extra information the project description has been added in appendix A and the overview of the SCRUM sprint results can be found in appendix B. Finally, Appendix C contains the report written at the end of the research phase.

3
Methodology

This chapter will cover the methodology of this project. The chapter starts with the development strategy in section 3.1. After this a general planning is shown in section 3.2. Next the used tools, software and programming languages are described in section 3.3. The chapter concludes with information about the documentation in section 3.4.

3.1. Strategy

One of the requirements of the bachelor project course is using a proper development methodology. For this project a minimized Scrum has been used as development methodology.

Agile development comes down to developing in an iterative process. An Agile development methodology means that it promotes adaptive planning, ensures delivering working software frequently, delivers continuous improvement and encourages rapid response to change (of requirements)[2] . Scrum is a type of agile development and therefore has these benefits.

Scrum consists of sprints, which are a basic unit of development, often one or two weeks. Each of these sprints begins with a meeting to plan the work to be done that sprint. During this meeting, tasks for the sprint are identified and are given an estimation of the time it will cost to finish it. During a sprint, the development team creates the functionality assigned to them. Each sprint ends with a retrospective meeting in which the process of that sprint is reviewed, allowing the team not to make the same mistakes during the next sprints. This causes the team to run a full development cycle during each sprint, including planning, requirement analysis, design, code creating and testing. During the full project, only completely working and tested functions should be integrated to the project, assuring there is always a working copy available.

During a sprint a short daily meeting is held to receive and provide feedback on the sprint. This includes discussing what has been done the day before, what is planned to be done this day and what problems could slow down the development. These meetings should be kept short, around fifteen minutes.

A Scrum team in general consists of multiple standard roles. These roles include:

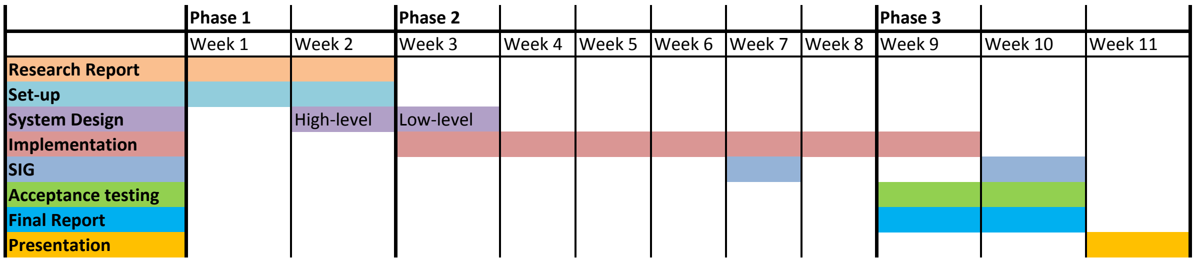
- the Product Owner, which is responsible for determining the work that should be done for the product and prioritize it on business value. He also communicates with the stakeholders. This work sorted on priority is written down in the *Product Backlog*
- the Development team, which produces the code and tests individually. They discuss the work to be done within a sprint with the Product Owner and the team mostly consists of three to nine people.
- the Scrum Master, responsible for adhering to the Scrum methodology, enforcing time limits on meetings to not unnecessarily distract the development team and the scrum master facilitates the Scrum process in general.

Since the development team consists of two persons, following the usual Scrum methodology would cause much overhead. The underlying principles can be applied though. During educational lunch lectures provided by the Delft University of Technology, several guest speakers told: "The best way to use Scrum is by doing it in the way it works for you". Therefore we did not use the standard roles and both of the team members adhered to each role. The labor therefore was divided over the team members equally.

During the project it was decided to do weekly sprints, due to the relatively short development time. The sprint meeting has been done at the beginning of the week and the plan for that sprint was logged for the client, coach and other members of the TRADR consortium. These plannings are included in this report in appendix B.

3.2. GENERAL PLANNING

At the start of the project a project plan was made which contained the high level planning. This can be seen in figure 3.1. The project is divided into three main parts; a research phase, a development phase and a concluding phase. This separation is also visible in this report where each phase is covered by a chapter.



	Phase 1		Phase 2						Phase 3		
	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11
Research Report	🟧	🟧									
Set-up	🟦	🟦									
System Design		High-level	Low-level								
Implementation			🟥	🟥	🟥	🟥	🟥	🟥	🟥		
SIG							🟦			🟦	
Acceptance testing									🟩	🟩	
Final Report									🟦	🟦	
Presentation											🟨

Figure 3.1: General overview of the planning from project plan

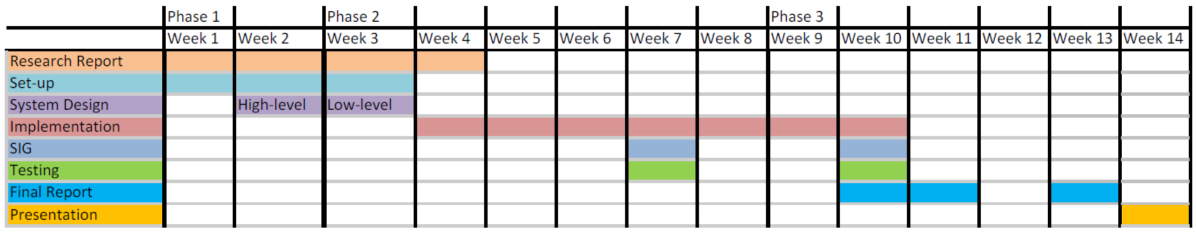
The actual work is represented in figure 3.2. The differences between the original and actual planning are caused by:

1. Missing access to documents in the research phase. The TRADR project has a wiki in which all information is gathered but it took a week to get access. In the meantime, general research was done on modules and other software to use.

2. Failing software installations in the research phase. A guide and setup script was provided to install all existing software but the installation of this failed. A lot of software would not load or compile as it should. This was solved when going to Fraunhofer, Germany, in week three of the project. Here two days were spent on fixing the software issues, finalizing the requirements, gathering information and making a code base to start from.

3. Because of all the delays the research report was not finished at this point. This was finished before starting the development phase.

4. Because of the late implementation of the database there was little time to finish this report on time. This is why the deadline has been extended and the next possibility for everyone to attend was weeks later.



	Phase 1		Phase 2						Phase 3					
	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Week 14
Research Report	🟧	🟧	🟧	🟧										
Set-up	🟦	🟦	🟦											
System Design		High-level	Low-level											
Implementation				🟥	🟥	🟥	🟥	🟥	🟥	🟥				
SIG							🟦			🟦				
Testing							🟩			🟩				
Final Report										🟦	🟦		🟦	
Presentation														🟨

Figure 3.2: General overview of actual work

3.3. TOOLS AND SOFTWARE

Since the TRADR project is in its second year at the time of writing, some decisions have already been made about which software, frameworks and languages to use. This shaped the context in which the project was performed, as the team had to adhere to the choices made by the TRADR consortium. Experiences with a previous version of the TDS had also to be taken into account. In this section the description and reasons for each used piece of software is given.

3.3.1. PROGRAMMING LANGUAGE

The MapView needs to be a plugin for RQT. RQT is the visualization plugin for Robot Operating System (ROS).[3] ROS is being used for example for communicating and operating the robots.

An RQT plugin can be written in a small set of languages,[4] in particular C++ and Python. Both languages have their advantages and disadvantages. C++ is a language which requires a fair amount of micromanagement of the memory, whereas Python is a relatively simple language. Given the fact that both team members had little experience in either of those languages, a language with an easier learning curve allows the team to develop more efficiently, especially during the first weeks of the development phase.

Both Qt[5] (discussed in section 3.3.2) and Marble[6] (discussed in section 3.3.4) are written in C++. To use a C++ library in Python code, there is a need for bindings. These bindings allow calls to C++ code using Python-like calls, therefore not requiring any knowledge about C++ if the bindings already exist. Therefore writing the plugin in C++ will require no bindings, but only require the libraries. This is a big advantage over Python, which requires bindings to use other programs such as Marble and Qt. These bindings can possibly be incomplete in comparison to the original C++ source code. Missing bindings can lead to problems if the bindings are required for important functionality as can be read in section 5.3.

It is recommended to write RQT plugins in Python according to the wiki-page about RQT[7] due to the ease of maintaining the plugin. It is also stated that C++ is accepted if the developers are more comfortable with C++ than Python or if there is a need to use libraries unavailable in Python. Since Qt and Marble both have bindings and the team was not comfortable with C++, these conditions were not met. Previous code written for the TDS has been done in Python as well. Therefore it is possible to use this code and integrate it into the TDS easily.

The choice for Python as programming language has been on the grounds of the expectation on how quick it was to learn to use properly and that it is recommended to use for RQT plugins. Getting accustomed to C++ can take a long time and could therefore slow down development. Since Python bindings were available for all required libraries, it was possible to use it without expecting problems.

With the choice of Python, the choice between versions came up. There are two broadly used versions of Python at this time; Python 2 and Python 3.[8] Python 3 is seen as the go-to version nowadays, but Python 2 is still standard with the operating system Ubuntu 14.04 and is the standard in the TRADR project.

For writing the TDS, the choice was made to use Python 2. This was due to compatibility with libraries such as Marble. These libraries require the use of Python 2, as the bindings for Python 3 are not stable or not available yet. The use of Python 2 instead of Python 3 should not have any effect on the development of the TDS. Both languages are of high quality and do not differ much from each other. They are also well documented and do not have a direct effect on the planned structure of the program.

3.3.2. GUI FRAMEWORK

As the main objective of the creating the TDS is creating a proper Graphical User Interface, a framework for building this GUI is important. This framework should at least allow a modular setup of the interface and should be able to work with the provided MarbleWidget.

It should also be compatible with RQT to be able to fulfill the requirements. RQT has been built using Qt as base as the name might suggest. More about RQT can be found in section 3.3.4. Marble's widget for showing the map with its controls is written with Qt as framework for the GUI. Given that both Marble and RQT require the plugin to be written in Qt made the choice for a GUI framework much easier.

Qt is multi-platform, therefore making it able to run the GUI of the TDS on multiple different platforms. The TDS will mainly be used on Ubuntu 14.04, but there is ambition to make it able to run independently of the platform. Qt makes sure that the graphical element of the TDS is usable on these different platforms, including tablets and smartphones.

There are multiple versions of Qt available at the moment, Qt4 and Qt5. Qt5 features many new possibilities and fixes for many bugs in Qt4. However since RQT still only supports Qt4, it was decided to use Qt4 during the development of the TDS. Using this older version should not create any problems, even though some more advanced functionality might be unavailable.

3.3.3. DEVELOPER TOOLS

The following tools are used for developing the product:

1. Pycharm:[9] This intelligent IDE (integrated development environment) provides all the functionality expected from a decent IDE but also provides code analysis tools for better code. It offers an educational license and is also able to use a remote interpreter. This was needed because one of the developers worked in Windows, while the product is not running on that platform.

2. Qt Creator/Designer:[10] Visual editor for GUI's in QT which can generate .ui files. These files can be read by QT and generate a user interface. Although this program offers more functionality, it was in this project only used to visually make an interface and generate the code for it. It was also used to generate a resource file containing all icons in the program.

3. Sourcetree:[11] A git tool which shows a clear overview off all branches.

4. Trello:[12] Ticket and other general purpose planning system. Used for notes, Scrum and maintaining the status on every task.

5. Redmine:[13] Ticket system of TRADR, which also hosts a wiki system containing most of the relevant information on the TRADR project.

6. Gitlab:[14] TRADR's git system. All our code is in this versioning system.

7. Overleaf:[15] Online LateX editor for working online on tex document with multiple people at the same time. Used for producing all documents.

8. PyUnit:[16] Python's standard unit testing library.

9. Sphinx:[17] Application generating documentation from dostrings in the source code. It is an requirement of the project. The requirements also state that Rosdoc has to be used for documentation of ROS packages, though this is not done because no ROS packages are used for the TDS.

3.3.4. OTHER SOFTWARE, LANGUAGES AND FRAMEWORKS

Next to above developer tools is other software where the product is built upon.

1. Marble: Marble is an application that can display a view of the earth.[6] It can be used as an application of its own, but as it is an open source project, it also serves as an example of how to use Marble widgets in a self-made program. A big advantage of Marble is that is runs on every operating system that supports Qt4, which is also used in the TRADR project. The reasons for using Qt4 instead of the newer Qt5 are described in subsection 3.3.2.

   In order for Marble to work, some dependencies have to be installed. On a clean install of Ubuntu 14.04, the following packages are needed:

   - SIP: A tool needed to create Python bindings for C and C++ libraries. In this case the PyQT library.
   - PyQT: A set of Python v2 and v3 bindings for Qt. See below.
   - PyKDE: Provides the python bindings for KDE, where Marble is a part of.

   Together they combine all the advantages of KDE, Qt and Python.

2. ROS: ROS is a framework which is used to make it easier to program robots. It is being used for the TRADR project, and needs to communicate with the TDS. This is because ROS sends information between its processes by sending messages, also to and from the TDS. In the current TRADR project, ROS Indigo is being used, which is not the newest version of ROS. Though, Indigo is the most stable version at the moment and recommended for stability.[3]

3. RQT: RQT is the base on which the program has to be build. It is required for the product as ROS is being used by the TRADR system as base. RQT is a plugin for ROS which allows developers to build a GUI on the ROS platform in the form of plugins.[18] These plugins can be written in Python and C++, in combination with Qt. The plugins written for RQT can be used as a standalone program, or as a window docked inside of the RQT window. As told before; the TDS consists of multiple separated parts; the OCU

and the MapView. These two parts may be used together at the same time, on the same machine. To make it easy to use these programs together at the same time, RQT allows arranging multiple plugins around the window by clicking and dragging. Due to the fact that this feature is already built in, the program is built to be embedded in the RQT window.

4. PyQT: PyQt is a set of Python v2 and v3 bindings for the Qt application framework and runs on all platforms supported by Qt including Windows, MacOS/X and Linux. PyQt brings together the Qt C++ cross-platform application framework and the cross-platform interpreted language Python. PyQt is able to generate Python code from Qt Designer. PyQt combines all the advantages of Qt and Python. A programmer has all the power of Qt, but is able to exploit it with the simplicity of Python.[19]

5. Untangle: Converts an XML-string or file to a Python object. Used for communication with the databases.[20]

3.4. DOCUMENTATION

It was required that the code was properly documented during the development. The documentation is especially important as further changes and additions to the program will need to be done as the TRADR project continues and other developers will need to understand the source code of the program to be able to work with it. The documentation was written using Python dostrings and there is also a separate guide for installing the software.

The Python docstring are interpreted by the autodoc feature provided by Sphinx. These docstrings were written following the Sphinx autodoc coding standards.[21] This allows Sphinx to format the string to highlight descriptions of parameters or return values. The documentation will be parsed into HyperText Markup Language (HTML).

Next to this, a setup guide has been provided with the application. This report, with requirements and design will also be uploaded to the TRADR project wiki. On this wiki other information is available such as use cases and older schematics.

4

Research Phase

The project started with a research phase. In this phase the optimal solution is researched by gathering and comparatively analyzing the necessary background information. In the contents of this chapter the given problem will be described and analyzed. The chapter starts with the problem definition in section 4.1. Analyzing the problem is done by presenting user and functional requirements in section 4.2. The chapter ends with a comparison between the envisioned GUI and the actual implemented GUI in section 4.3. Here is also the answer for the second research sub question how to effectively display the information needed by rescuers.

4.1. Problem definition

The TRADR project is an EU-funded project developing technology for human-robot teams to assist in disaster response efforts. This is done using a proven-in-practice user-centric design methodology, proven in the NIFTi project.[22] In the TRADR-scenario, robots collaborate with humans to explore environments and gather samples. Through these scenarios, the team develops knowledge of the disaster area over multiple -possibly asynchronous- missions with as goals, improving team member's understanding of how to work in the disaster area and how to improve team-work. The environments these scenarios take place are medium to large scale industrial accidents, with mission times possibly stretching over multiple days. Teams consist of human rescuers, UAVs and UGVs.

The TRADR wiki provides in an example scenario on which also the product produced in this project will be tested. The general scenario of the second year is as follows:

Highly flammable Xylene has caught fire, the fire seems small but can not be put out by the company firemen. Due to the heat a pipeline filled with resin breaks and more and more resin is pumped onto the flames. This fire is handled wrongly by the employees causing a spread of the fire in the direction of Intermediate Bulk Containers (IBC's) which melt within a minute causing their contents to spill with as a result a non-containable fire. When the firebrigade arrives it is not sure if all employees have left the premises, the toxic fume spread is not known and there is still highly explosive ethylacetate on the premises.

During a TRADR-scenario, the data from the robot is filtered and (if needed) approved by its operator and gets sent to a set of databases. The low-level database is a MongoDB[23] database to save documents, photos and other data such as sensor data from the robots. All entries in this database are time-stamped to allow possible changes in the situation to be visible.

The high-level database is a StarDog database. This database makes use of an ontology to allow reasoning between "Things" or schemes. An ontology is a representation of the data (knowledge) by using schemes within a domain and the relationship between these schemes. These relationships can be used to for example categorize data automatically. This can be the fact that a fire is a hazard. Since the TRADR project aims at human-robot teaming, this kind of reasoning is important to allow persistent situation awareness, allowing the robots to make sense of possibly conflicting information and to allow the robots to determine interactive behavior for use in team-level coordination.

During an USAR mission using the developed technologies in the TRADR project, it is critical for the rescuers that the data retrieved using the robots is shared and displayed in a clear manner. Rescuers should therefore be able to access and interpret the data quickly, without costing much time or energy. An option to communicate the information to the teams is by using portable devices such as tablets, and visualizing the required information on their screens. By using such a user interface, rescuers in the area can quickly see hazards and other information on a top-down map. It is also possible for the leader of the USAR mission to plan strategies dynamically using such an interface and sending his/her commands to the teams.

The TRADR project is evaluated multiple times during a TRADR Evaluation. During the evaluations the systems are tested during a simulated USAR mission. During these evaluations multiple GUIs have been used.

Before this project, there was a UI which consisted of a browser environment with a top-down satellite view of the mission area. It also had the possibility to track actors and to add text and images, retrieved from the previously mentioned databases. It was built on top of the Google Maps API. This UI was a pragmatic approach, because there needed to be a GUI at TRADR Joint Exercise (TJEx) and there was none at the time.[24]

Since the project is still in heavy development and more functions are being built to assess areas, an interface without the possibility of easy expansion is not a workable option and it costs the team valuable time. Therefore a new user interface (the TDS) is needed, which is built in such a way that it is easy to expand with for example plugins. The OCU is a GUI created for the robot operators, allowing them to control a robot and receive direct information such as video streams from a camera in the robot. A global overview of the top level TRADR architecture can be seen in figure 4.1. The information streams received in the OCU contain much information that is not important for all team members, such as video footage of a robot moving to its goal.

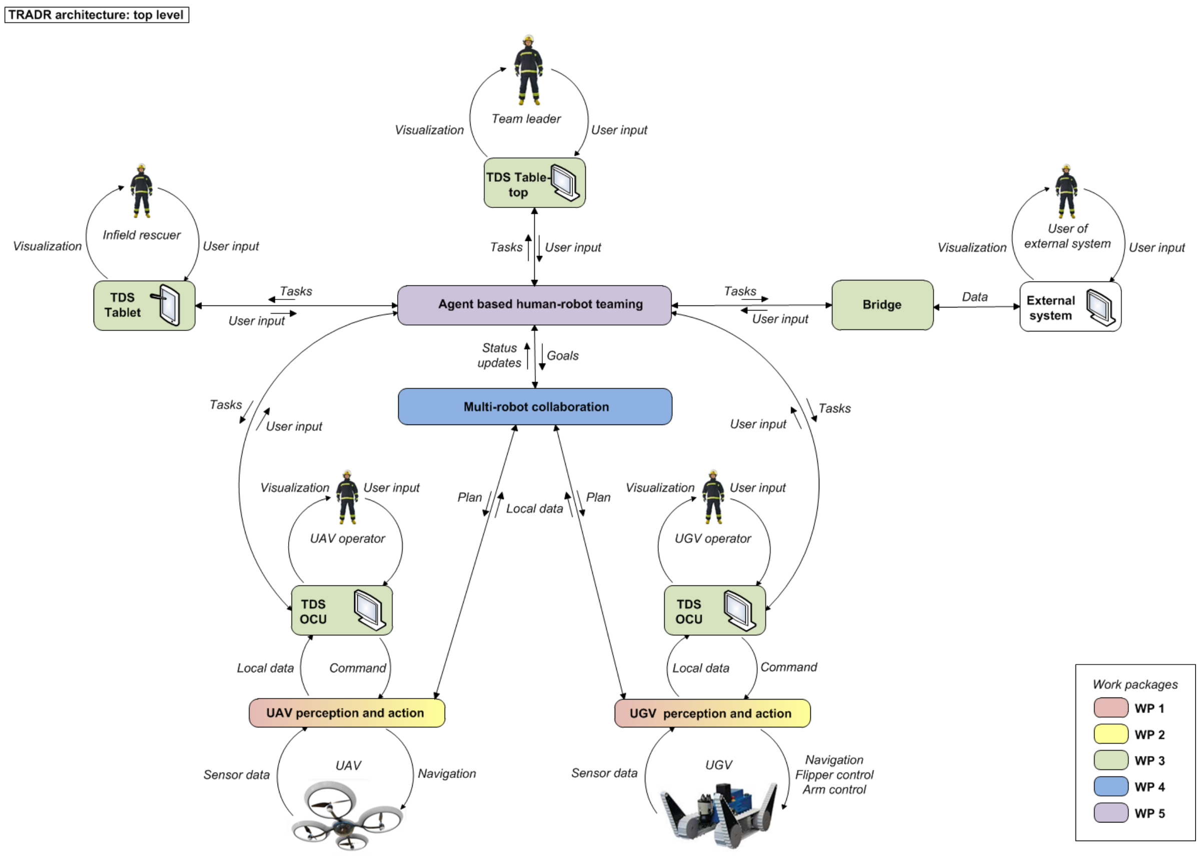


Figure 4.1: Top level TRADR architecture where the position of the TDS can be seen as part of a larger system. *Source: (Private) TRADR wiki; URL: https://redmine.ciirc.cvut.cz/projects/tradr/wiki/System_Architecture. Retrieved on 16-8-2015*

4.2. PROBLEM ANALYSIS

During the first few weeks a list of requirements has been set up in collaboration with the client. These requirements are based on the problems raised in the problem definition. All requirements found on the projects Wiki - which were scattered as they cover feedback from multiple years of work - were gathered together, categorized (see next paragraph), and then the requirements were talked through with several TRADR consortium members and adjusted accordingly. These members are the TU Delft, TNO and Fraunhofer. After the last TJEx, at which the pragmatic version of the TDS had been used, feedback has been given by the end-users. This feedback has been taken into consideration while creating the requirements.

The requirements are split in multiple types. Functional requirements, non-functional requirements and user requirements. Functional requirements are requirements referring to the functionality of the program. Non-functional requirements refer to how a program should perform and for example what system it should run on. See also appendix C for the non-functional requirements in this project. The user requirements are a description of the different users and how and why they use it.

4.2.1. User Requirements

The users of the TDS are mainly firefighters during urban S&R operations. Due to the chaotic situation during such an operation, it is important that the TDS does not require much knowledge to use. All actions need to be intuitive to all users. The users are defined as follows (the list is not exhaustive):

The *Team Leader* leads the mission and coordinates the multiple teams. His main tool to keep an overview will be the MapView. This means the overview should stay orderly when more information is added to it and it does not get cluttered.

The *Operator* will mainly use the OCU part of the TDS, and uses the MapView on the side. Therefore the MapView needs to be able to run alongside the OCU without the MapView interfering too much with the GUI of the OCU.

The *In-field Rescuers* are the persons who explore the mission area. Their tasks include assessing the situation and sending information and observations to all other devices. If the in-field-rescuers will use the TDS, they will only use it in read only mode in the scope of this project. In the future there is a possibility that in-field-rescuers will add information to the MapView and the database. This is not a functionality covered in this project as the in-field-rescuers will use an adapted UI on some portable device for this.

4.2.2. Functional Requirements

The functional requirements are structured using the MoSCoW principle, splitting it in must-have, should-have, could have and won't-have requirements. These requirements have been made at the beginning of the project (see also appendix C). Since then it has been slightly revised due to new planning of time. Below are the revised requirements. If a requirement is in bold this means that it is implemented in the final product.

Must Have

The product must have:

1. **a modular set-up of interface: be able to show or hide certain components and personalize interface according to role, personal preference and device**
2. **a layered structure (where layers can be switched on or off) with**
    - **base map**
    - **location of team members**
    - **info icons (Points of interest such as victim, fire, hazardous material, obstacle, remark, photo)**
3. **info rights management (who can see what, based on different roles)**
4. **give a notification in case of faulty connectivity and reestablishes it when available**
5. **display of photos**
6. **a structure allowing plugins to extend the program**
7. **the possibility to run multiple instances on different machines, while displaying the same information on the multiple instances**
8. **a setup guide**
9. **handheld device support: touch control and keyboard on screen when using a touch screen**
10. **user friendly feedback when an error occurs.**

Also, the product must:

- **run on Linux**
- **use KDE Marble**

- **be implemented as a set of RQT plugins**
- **use databases as ground truth for data retrieval and storage, in particular the high-level Stardog (RDF) and low level MongoDB database from the existing system.**
- **be fully documented using Sphinx and Rosdoc**

SHOULD HAVE

The product should have:

1. the following added layers:
   - layer for photos, including clustering where multiple pictures have been taken (like in Google Maps)
   - layer for robot trajectory traces
   - area indication ("working agreements") (go, no-go, unstable, searched, robot autonomy)
   - user input on the map (Text, notes, drawing, photos)
2. the following icon features:
   - icons should be informative (e.g. thumbnail icons from photos, indication of robot control)
   - **icons should scale naturally**
   - possibility to lock icons (as to not move them by accident, possibly default)
   - show field of view/orientation for photos and robots when data is available
3. the following properties for information in the database:
   - **timestamp of entry**
   - name of actor who entered the information
   - **visible in TDS**
4. **an indicator for reachability of actors**
5. **shortcuts to locations**
6. map tools (**scale info**, distance measure)
7. **GPS-coordinates visible within TDS (when adding icons and current view)**
8. *(moved from must haves)* a communication ability with other ROS nodes (e.g. to display a video stream)

COULD HAVE

The product could have:

1. Robot status info screen (battery, connection status etc.)
2. time line of events (possibly within main TDS for team leader)
3. possibility for situation report or briefing
4. **representation of team management information (e.g. showing team members; show goals, tasks and status)**
5. global search function (e.g. search for data, text, objects)
6. Streetview-like option to view specific area as 3D map (Overlay the 3D point-cloud map with the 2D plan)
7. information from external sources (e.g. weather information)
8. possibility to show videos and play back videos
9. speech input (in cooperation with DFKI)
10. display of (co-)variance of GPS location
11. comment functionality on photos and other entities
12. simple image editor
13. **multitouch functionality (gestures, pinch-to-zoom)**
14. recovery options (e.g. using the last position of a robot when the connection is lost or power supply is not functioning)

4.2.3. DEFINITION OF DONE

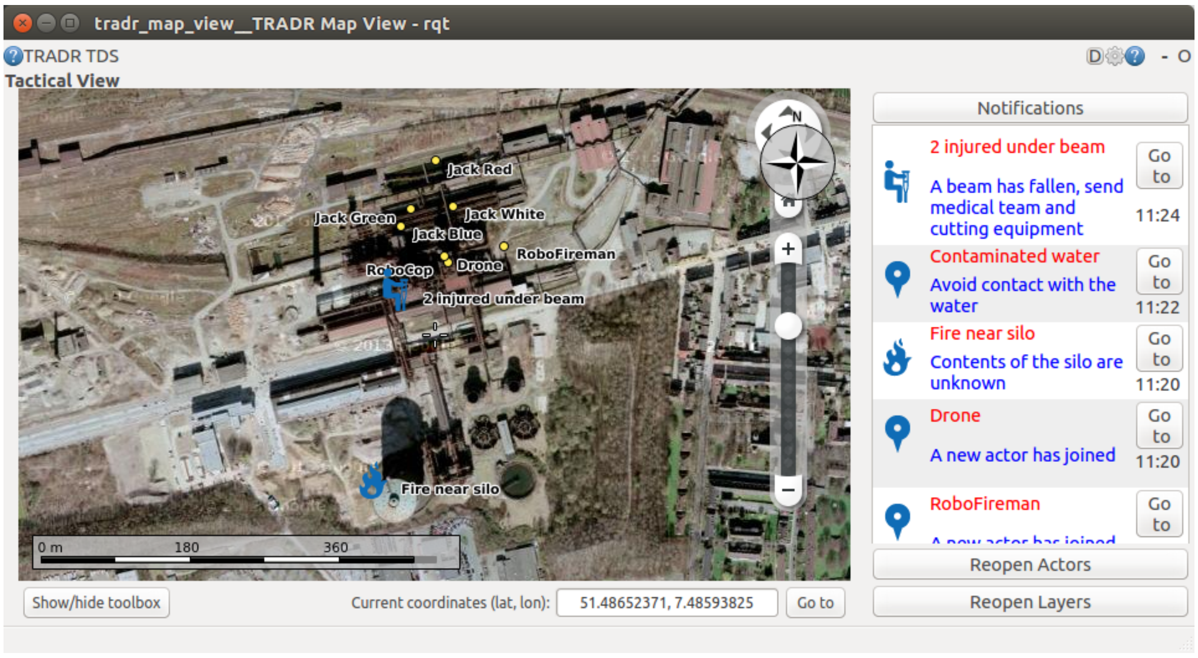
This subsection will state the success criteria which the finished product has to meet to be called done. For the product to be finished, the system has to:

- have all *must have* requirements from subsection 4.2.2 implemented
- pass all acceptance tests
- have no high priority, severe or program breaking bugs left outstanding.
- run on the required system as stated in subsection 4.2.2

4.3. USER INTERFACE DESIGN

In this section the design of the TDS will be discussed. The actual GUI looks a lot like the envisioned GUI in the research report (see appendix C). The interface is divided in three main parts; a left sidebar containing tools for map interaction, a main section where the map and potential error messages are displayed, and a right sidebar containing information on the mission.

During development of the GUI of the TDS, multiple facets were taken into account. Guidelines for the design of the OCU have already been defined beforehand, unlike the TDS design. Therefore the design of the MapView has been made at own accord. Information for this was gathered from multiple team members of TRADR which told about their own experience and feedback from users in the field. Also information from the TRADR wiki was used as also photos from the previous GUI of the TDS.



2 injured under beam
A beam has fallen, send medical team and cutting equipment
Go to
11:24

Contaminated water
Avoid contact with the water
Go to
11:22

Fire near silo
Contents of the silo are unknown
Go to
11:20

Drone
A new actor has joined
Go to
11:20

RoboFireman
Go to

Figure 4.2: Interface of the TDS with multiple points of interest on the map, including actors. This interface is sized to a smaller screen size such as a tablet and parts are minimized to save space

The application should first of all not distract the user, except for important error messages. The most important task of for example the mission commander is keeping an overview of the situation, and thus the map of the area. This is the reason the map is the center point of the application. When there is a system failure the application displays an error message above the map. An error can be caused by for example either the user by entering an illegal coordinate, or the system when the connection to the database is lost. It is then still possible to use the rest of the program while fixing the issue which caused the problem. An example of a displayed error message can be seen in figure 5.7.

Another example of non-invasive messages is the notification center. Located in the top of the right sidebar by default, the notification center displays messages when new information is available in the database. This can be a range of messages, as for example a new POI, actor, photo or for future functionality also notes or videos. The icon within the message is descriptive of the type of message. This way it is possible to visually filter the messages and know in an instant what type of information is dealt with.

The second facet taking into account during developing was that it would also be used on a portable device such as a tablet-pc. These devices often have smaller screens than a laptop or desktop monitor. This can cause text or other information to be too small or cramped on the screen to keep an efficient overview. Since it is very important to keep the shown information organized and clear, the design of the MapView should allow the user to turn on and off parts of the interface. To do this, the MapView-plugin lets the user minimize parts of the interface such as the toolbars. The user can focus on the functionality he needs and hide the functionality not required at the moment, giving the user control of the information shown. An example of the application running on a tablet can be seen in figure 4.2. Here the toolbar and two information views have been minimized to save space.

During the research phase it was found (see Appendix C, section 2.5) that with the optimal level of transparency, the optimal division between divided and focused attention can be achieved.[25] The paper by Harrison researches how one can better support both focusing attention on a single interface object (without distraction from other objects) and dividing or time sharing attention between multiple objects (to preserve context or awareness). Special attention is paid on how effective the use of transparency is in combination with small sized screens and systems with many windows, menus, tools, etc. The conclusion of the paper is that it is useful for certain user interface elements to use transparency for a better divided attention. This is relevant for the GUI in the current research as there is a need to focus on a specific situation during an USAR mission, while preserving awareness on the complete mission. This however is not implemented in the final product. As the view of the map should not be obstructed in any way, as requested by the client during a meeting, is transparency not useful for the TDS. Other elements in the UI did not have an apparent reason to have transparency and it is thus not implemented.

5 DEVELOPMENT PHASE

After four weeks the project reached the development phase. Here actual code was produced and the program was build part by part. In this chapter can be read how the software has been built and what problems were encountered during the development.

The development phase started with making a plugin for RQT which can be read in section 5.1, together with other implementation details and the answer to the first research sub question: *How to make a modular GUI*. Next an overview of the implemented system is given in section 5.2. During the development phase several problems were encountered which is elaborated in section 5.3.

5.1. GENERAL IMPLEMENTATION

This section contains general information about the implementation. This includes a description of how to make a plugin for RQT, how to make the plugin modular and the used design pattern.

5.1.1. MAKING A PLUGIN

Instead of making a standalone program it was required to build a plugin. TRADR uses robots during a mission which run on an operation system called ROS.[3] Within the TDS are no connections to the robots itself, as all data transfers are being done by the database. It would still be very useful to incorporate the TDS in the software for controlling robots and other TRADR related programs. This is why TRADR uses RQT, which is a Qt based framework for GUI development for ROS.[18]

As a result, the TDS has been built as plugin for RQt. This means that a person can simultaneously have a look at the TDS for an overview while in the same window giving commands to a UAV, as most RQt plugins can be arranged however the user desires. This also means that much of the functionality can already be available at the startup by sharing common procedures like starting and closing the application. Common procedures include closing the database connection and closing files that are in use during shutdown.

5.1.2. BUILDING WITH PLUGINS FOR MODULARITY

The application can be launched from within RQT after which the program starts to initialize. As modularity was important to the client, the decision was made to divide the code of the application into the base functionality and extending it with plugins to add more functionality. The architectural pattern of the code base is based on a Model-View-Presenter (see subsection 5.1.3). This division of concerns is implemented in the initialization by saving all models, views and presenters together in separate lists located in one place.

A general 'add plugin' function makes it easier for other developers to add their own plugin to the program. This can be as simple as giving a name and passing the model, view and presenter of the plugin. As not every plugin needs its own model or view they can be given as parameter to other presenters. Finally, after initializing the database and configuration settings, a loop is started in which every plugin in the given order is activated. Each of these plugins initializes itself after each other. The given order makes sure dependencies are met.

5.1.3. MODEL VIEW PRESENTER

There are many possible different design patterns to follow when creating a program. This can be done to for example increase the ease of maintaining and extending the program.

The plugin has been built in a model-view-presenter (MVP) pattern. This approach was chosen as this well-known pattern makes it possible to change individual parts of the program. In the MVP pattern there's a separation between the knowledge (Model) the UI (View) and the logic (Presenter).

All logic for the UI is available in the presenter, which communicates with the view through an interface containing the base functionality for the view. The actions raised by user input will call to the view, which passes the information to the presenter. The presenter then performs all logic including the logic and tells the view what to change.

The pattern consists of creating an interface or base class for the model and the view. By doing this, the model and view are loosely coupled to the presenter, making it easier to change or mock the view and model. This is useful for making unit-testing easier for example. The functions defined in the base class are required to be implemented for the class to work with the basic functionality.

Within the presenter class, no code will be adjusted when a model or view changes, only added. Therefore by using this pattern, the system design follows the Open-Closed Principle.[26] By following this principle old functionality is almost guaranteed to stay intact after system changes. This allows relatively easy addition of functionality, therefore keeping the software maintainable and extendable.

The MVP pattern is a derivation from the Model-View-Controller (MVC) pattern. The MVC pattern uses a Controller instead of a Presenter. This controller received all user-input directly and returns a view to be rendered. Because of this, the view is only required to render itself. An example of this is visible in figure 5.1

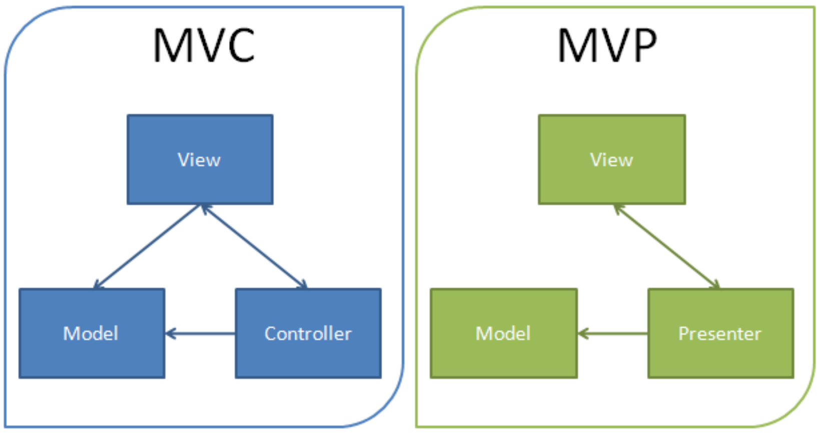


Figure 5.1: The difference between the MVC pattern and the MVP pattern. Image retrieved from: http://stackoverflow.com/questions/1317693/what-is-model-view-presenter on 27th of August 2015

Since the GUI is created using Qt (see 5.2.3), using signals sent from the view and received in the presenter is an easy and efficient way of responding to events raised by user input or input from the database. As the program would need to be tested too, the MVP pattern has the benefits of using the interface for the view. This can be mocked much easier than by using a MVC pattern.

Another benefit is that the presenter and view in an MVP pattern are often mapped 1-on-1. This creates a clear separation between multiple different functions within the TDS. This separation makes it easier to change certain functionality and to create plugins that extend only a part of an already existing function.

5.2. SYSTEM DESIGN

The design of the system has been created in the research phase. During the implementation, it was discovered that the chosen design did not provide in all necessary requirements, such as adding functionality to the program using plugins. Since this added functionality required changes in the system design of the TDS, the actual design differs greatly from the initial design. Differences between these two designs will be explained and the reason for these changes elaborated.

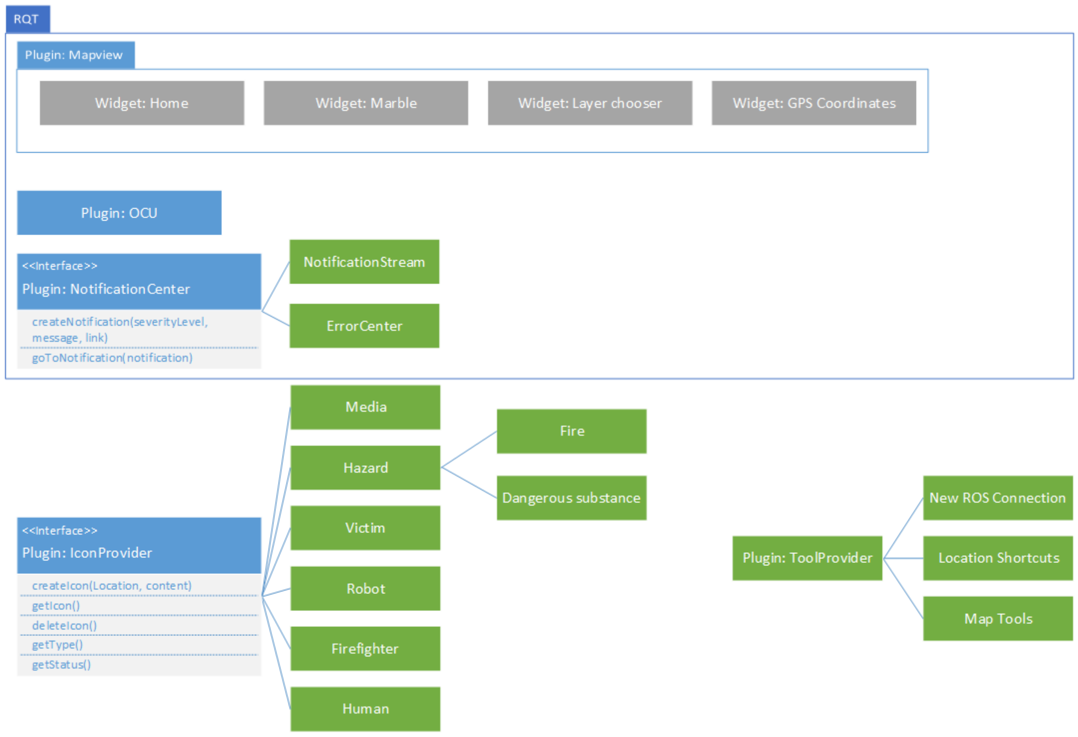


Figure 5.2: The initial high-level design of the TDS

5.2.1. INITIAL TDS DESIGN

The initial design consisted of multiple RQT plugins and some separate plugins which add functionality. A visual representation of the initial design is visible in figure 5.2

The most important plugin was the MapView plugin. This plugin was a plugin for RQT and consisted of the extreme base functionality of the system. It contained the widgets for showing the map, choosing which layers to display and showing the current GPS-coordinates. All this functionality is directly connected to the Marble widget and should therefore be contained in a single plugin. The already created OCU plugin for RQT would be able to run next to the MapView plugin and due to modular interface in RQT itself, it is possible to decide how to display these two.

The last plugin to be added as RQt plugin is the notification center plugin. The notification center will be responsible for showing non-intrusive error-messages and notifications, which could be of importance to the user of the TDS. This might include information about newly found hazards or the loss of connectivity. As the notification center will be required to send messages received from all plugins, for it to be a separate RQT plugin, it required to be able to connect to ROS-topics. These ROS-topics are the manner in which ROS sends messages between programs.

The initial design consisted of two different Qt plugins, not developed as RQt plugin. As these plugins do not require a direct connection to information from ROS-topics and their GUIs get added to the MapView GUI or require none, it has no advantages to add it as RQT plugin.

The first of these plugins was the IconProvider plugin. The IconProvider receives requests about points of interest. Requests could be for example adding a POI to the map, receiving all information saved in the POI or editing the information saved in a POI. The POIs themselves were supposed to be subclasses from the icon provider, each adding their own style or functionality to a POI.

The last designed plugin was the ToolProvider. This plugin was responsible of being the location where each tool was initialized and called. This central access point for the tools could provide the possibility to add more tools to the program, without requiring references to each of these tools. With a possibly growing set of required tools as the TRADR development continues, a general point of access makes it easier to add tools later on.

5.2.2. ACTUAL TDS DESIGN

The initial design did not allow the addition of plugins in a maintainable way. To add a plugin that required a connection to the Marble widget, it would need to be added to the MapView plugin. As this requires adapting the MapView plugin for each added plugin, it is possible to introduce bugs. It was also discovered that there was an interface ready for creating layers in the Marble Widget. To these layers, POIs can be added by using the placemarks (GeoDataPlacemark objects) from the Marble libraries. New and custom layers could be created by subclassing the LayerInterface class in the Marble libraries and adjusting its rendering functions.

During the creation of the new design in the development phase, the design got adapted to the Model-View-Presenter pattern. As this pattern created a separation of concerns which was not available in the initial design, many of the classes got split into a model, a view and a presenter.

The new system design also adds all widgets to the same RQT plugin. Adding everything to the same plugin causes only a single plugin to be loaded when starting the TDS in RQt It also allows code to be shared within these plugins. Later during the development process, it was discovered that sharing code with multiple RQT widgets was not a problem. Sadly this fact was discovered too late in the development process to easily adjust it.

A general visualization of the actual design is given in figure 5.3. The actual design consisted of several Python modules each with their own functionality. The MapViewPlugin class is responsible for instantiating the other plugins and adding them to one single widget to add to the RQt instance. This class is a subclass of the Plugin class defined in the RQT module. By subclassing this class and adjusting some methods, the plugin can be added to RQT and define a shutdown and startup procedure.

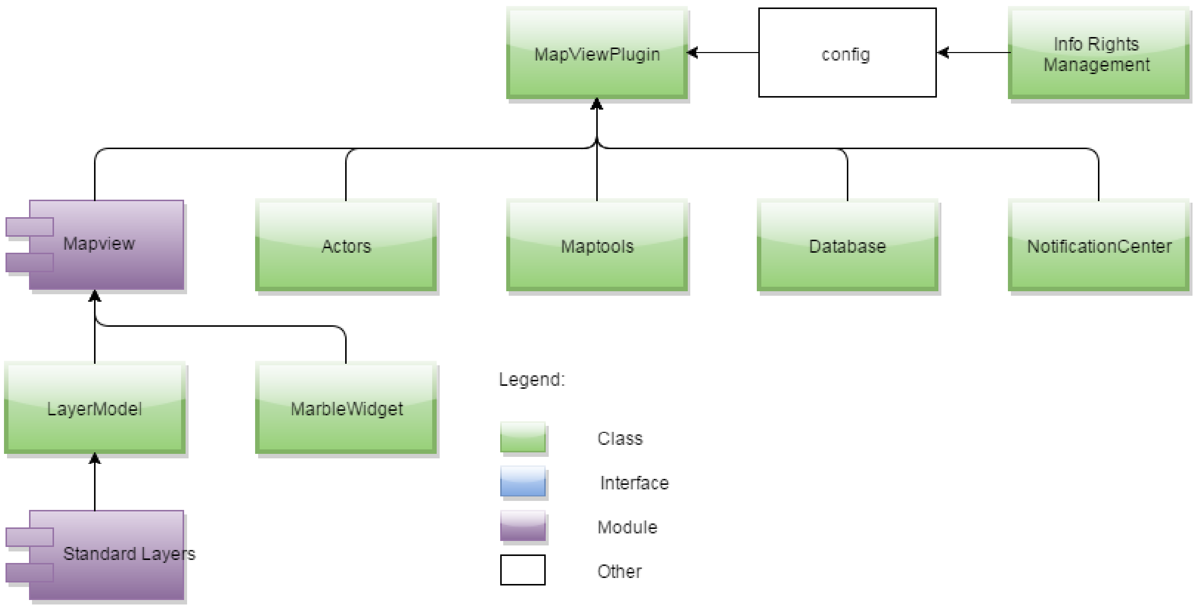


Figure 5.3: General overview of the actual design and implementation

MAPVIEWPLUGIN

The MapViewPlugin class contains references to all plugins which need to be instantiated when the MapView plugin starts. It contains several python dictionaries (a list accessible by a key instead of just an integer) containing references to all the initiated objects, to make them accessible for other plugins. This is required when a class in a different plugin requires a reference to an already instantiated view. This way, the required view can be found by calling the dictionary with the name of this view. In the current format, all plugins which create or need access to a toolbar require a reference to the MapView plugin. Also, every plugin requiring direct information from the MarbleWidget has a reference to the MapView, as the MarbleWidget is instantiated here.

The MapViewPlugin also creates a singleton object of the NotificationCenter class. The notification center should be accessible by every class which has to be able to show notifications or user friendly error messages. To make it accessible from every module, it is created using the Singleton design pattern. A singleton creates a single point of access to a created instance as static attribute of the class. The notification center has been implemented by wrapping the NotificationCenter class with another class. An instance of the inner notification center class is saved as a class variable of the outer class. This makes sure there is only a single instance of, and a single point of access to the notification center

The MapView is a module containing the model, view and presenter of one of the main components, the view of the map. The structure is visible in figure 5.4. The presenter acts as the middle man between the model and the view. It provides all logic for the view and retrieves this information from the model. The model and view are connected to the presenter through an interface. In the view class, the MarbleWidget which shows the top-down map is loaded, along with some other UI elements such as a collapsible toolbar.

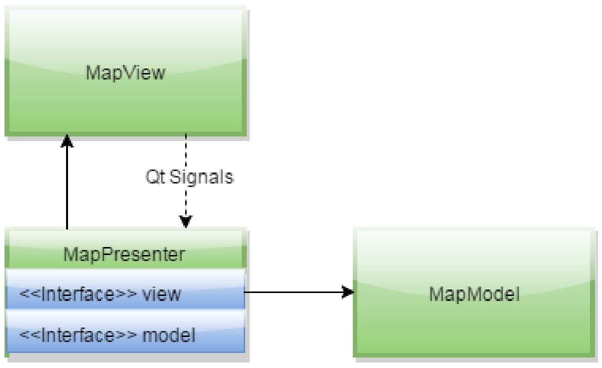


Figure 5.4: Overview of the structure within the MapView module

LAYERS

The LayerModel class is instantiated when creating the presenter class in the MapView module. This class provides a single point of access to the custom layers added to the MarbleWidget. It also provides the logic to add and retrieve layers.

The custom layers are created and saved in the standard_layers module, which is visualized in figure 5.5. These layers include a MarbleLayer class, which merely realizes the layer interface in the Marble library and adds standard functions for activating and deactivating the layer. This class has been subclassed by an empty layer class, providing no functionality. This empty class can be used as base for layers, which allow drawing or different renderings on the screen, not dependent on the placemark functionality in Marble.

The other class which extends the MarbleLayer class is the POI layer. This layer uses the placemark functionality from Marble to show placemarks on the map. Extending this class allows for different styles of renderings on the map. The already created list of layers is not exhaustive, and new layers can easily be added by creating a new class using either the MarbleLayer, empty layer or POI layer as base class.

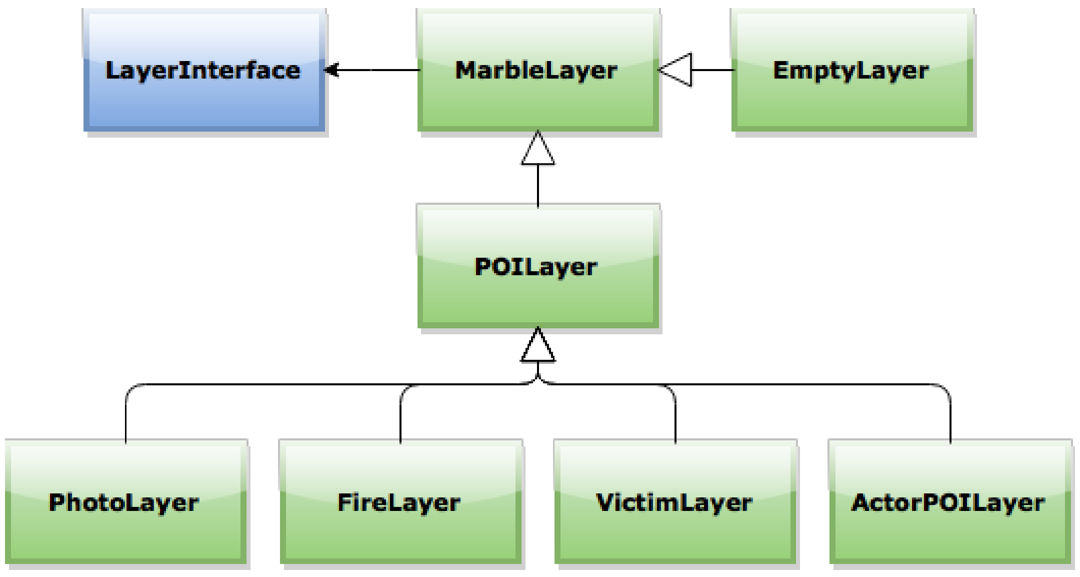


Figure 5.5: Stucture of the standard_layers module.

ACTORS

The actors are represented using the Actors module. This module contains a model, view and presenter for showing the actors on the map. The model contains a list of actor objects, containing all information of a specific actor. The presenter also initializes the view, which shows a toolbar. This toolbar shows all the information saved in the selected actor object.

To show the actors on the map, a new placemark is created in Marble. This placemark is shown on the actor layer. Since the coordinates of these actors change constantly and the current coordinate is saved in the database, a threaded class ('worker') has been created together with the actor layer in the standard_layers module. This has been placed here since this worker class is closely related to the layer for the actors, which is defined here. This actor worker polls in a defined rate to retrieve the new coordinates if available.

OTHER MODULES

The maptools module contains the standard tools that can be used to for example make navigating the map easier. This module has been created to facilitate all tools that are too small to require a separate module. Since most of the smaller tools are defined in the 'should haves' or 'could haves' of the requirements (section 4.2.2), the maptools module only contains the function to go to a saved location or to a custom coordinate in the current implementation.

A config module has been created to allow easier adjustment of variables that change in different setups, such as the database address or the currently used map theme. By adding all these variables to one single file, the variables can be adjusted much quicker by the user.

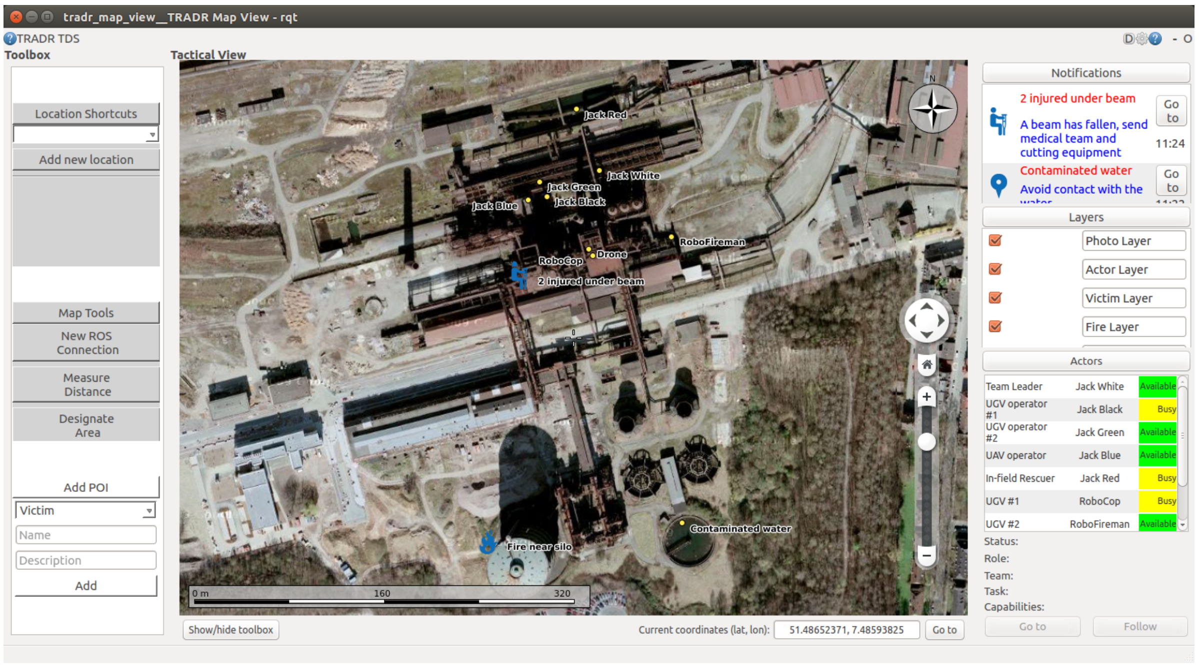
The config module has also been used as a point of access to functions that are required in many other modules and are not accessible using regular references. This can better be done by creating a new module containing these functions, instead of adding them to the config file. By doing this, the config would be adjustable in a more user friendly manner.

The info rights management class has been created to restrict functions from being executed by user who should not be able to do this. This allows for example to block certain groups from seeing pictures, which could contain privacy sensitive information.

The info rights management is implemented as individual functions in the config module. These functions are able to add a function to the list of restricted functions and to change the permissions each role has. To set up the restrictions, roles and permissions have been saved as enumerators. Enumerators work the same as a python dictionary, except that the keys are implemented as attributes of a class and it is not extendable during runtime. These enumerators can be extended or changed in the config file to change the permissions for each individual role. To restrict a function, an if-statement should be added to the beginning the function that has to be restricted, which breaks out of the function if the user is not allowed to execute it.

5.2.3. USER INTERFACE

The complete application is built as plugin and is internally also split into several plugins. This is reflected in the GUI which also consist of several 'plugins'. The GUI is written in Qt which allows to use widgets within other widgets. Because of this, the user interface consists of a base widget and multiple other widgets adding information to this base. These are all created using Qt creator which produces .ui files. These are XML-like files containing names and attributes for every button and frame available in the application.



Current coordinates (lat, lon): 51.48652371, 7.48593825

2 injured under beam
A beam has fallen, send medical team and cutting equipment
11:24

Contaminated water
Avoid contact with the water

Team Leader	Jack White	Available
UGV operator #1	Jack Black	Busy
UGV operator #2	Jack Green	Available
UAV operator	Jack Blue	Available
In-field Rescuer	Jack Red	Busy
UGV #1	RoboCop	Busy
UGV #2	RoboFireman	Available

Status:
Role:
Team:
Task:
Capabilities:

Figure 5.6: Interface of the TDS with multiple points of interest on the map, including actors. This interface is sized to a bigger screen size, such as a desktop or laptop with all information views open

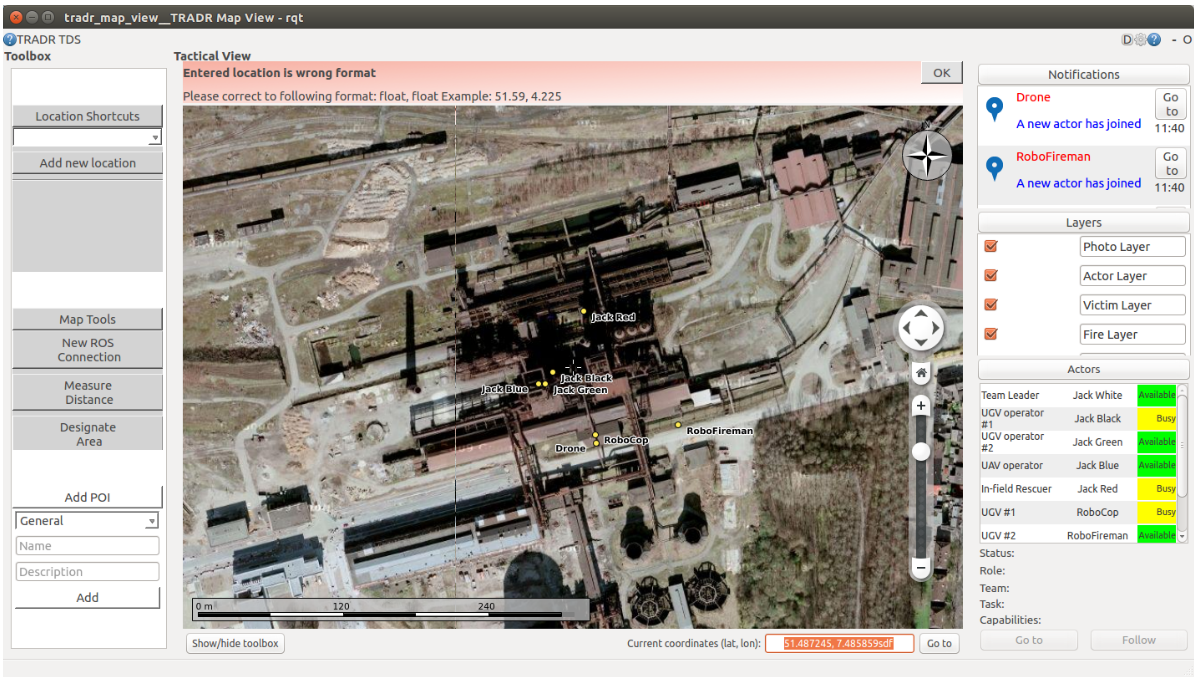
The base consists of a horizontal layout containing three parts; a left sidebar, center widget and a right sidebar. Only the left sidebar contains actual content which is a scroll area. This means that when many plugins (and its corresponding widgets) are added to this sidebar, it is possible to scroll or swipe down to access the other items in the sidebar, which normally would not be able to fit in it.

Widgets belonging to a plugin are loaded when the plugin is initialized. This means that when a plugin needs to change, only the accompanying .ui file needs to be altered. This modularity ensures easy maintenance of the widgets. Each widget is implemented as a QDockWidget to make it possible drag and drop the widget to a preferred location. This functionality did not work as expected, so this is replaced by a minimize function (see below). After loading the plugin, each element in the widget (e.g. buttons, text fields) is given a connection with a function. This is done in Qt by first connecting a signal (e.g. clicked, enter pressed, right mouse click) to a specific slot function, optionally accompanied with extra parameters as for example clicked location, or contents of a text field just entered by the user. The corresponding slot then gets executed.

This process of linking buttons with functions has to be done separately for touch events, generated by Qt when using a touch screen. As TRADR is going to use tablets and/or smartphones during missions, it was required to enable touch support. A 'touch click' in Qt is a completely different event than the standard 'mouse button pressed' event, which means that all connections have to be made manually. In Qt this can be solved by giving widgets (a button and textline is also regarded as a widget) an extra attribute which lets it accepts touch events as click events. This is why QWidget (the Qt container which can contain other Qwidgets) is subclassed to a custom TouchWidget class. The only difference is that an attribute is set during initialization. This *accept touch* attribute will be set for every child widget recursively.

To allow each widget to minimize, a special function available to QDockWidgets is used. This function replaces the standard title bar of the widget with a custom widget, which in the case of the TDS is only a QPushbutton with the original name of the widget in it. Clicking this QPushbutton hides the complete widget and it is then replaced with its own reopen button placed in the same position. After clicking on the reopen button again, the original widget is shown again and the reopen button hides again. This is implemented by extending each QDockWidget with a self-made QMinimizeWidget class which implements these functions. By doing this, this code only has to be implemented once and is easy to use for future authors of the TDS and maximizes maintainability.

The notification center, the layer chooser and the actor overview are the three current widgets in the right sidebar. These three widgets have a QListWidget (contained in a QDockWidget) in common. A QListWidget is a widget which allows multiple other widgets to be added and shown in a list format. Normally, the QListWidget displays items with simple lines of text. For the TDS this is adapted to allow widgets to be added instead of the regular text.



Entered location is wrong format

Please correct to following format: float, float Example: 51.59, 4.225

Drone
A new actor has joined
11:40

RoboFireman
A new actor has joined
11:40

Team Leader	Jack White	Available
UGV operator #1	Jack Black	Busy
UGV operator #2	Jack Green	Available
UAV operator	Jack Blue	Available
In-field Rescuer	Jack Red	Busy
UGV #1	RoboCop	Busy
UGV #2	RoboFireman	Available

Current coordinates (lat, lon): 51.487245, 7.485859sdf

Figure 5.7: Interface of the TDS with a non-invasive error message showing, caused by an incorrect input

5.3. ENCOUNTERED PROBLEMS

A lot of problems were encountered during the development phase. Some were program-breaking problems solved in the very last week, and other just took a long time to fix. Most of these problems arose from using Python as programming language. Marble, the main part of the system, is written in C++ and bindings are created to access its functions from the python environment. In the research phase however, it was unknown to the team that for some functions the bindings are not available.

5.3.1. PROBLEMS WITH PYTHON BINDINGS

For the program Python 2 was used in combination with Qt to provide the user interface. Also Marble was used as part of the program for the view of the top-down map. Both Qt and Marble are written in C++, therefore using these requires bindings to Python. This way Python can access the functions defined in the C++ code, granting the developer the possibility to use these functions in the Python code.

To make use of the features of Marble and Qt, libraries have been created which provide the required bindings. A big part of the functionality works by using these bindings. However within the PyKDE bindings for Marble, some features were missing. This was due to them requiring native C++ arguments or return types, or the fact that they were bugged and not stable enough to release.

INPUTHANDLER

These missing bindings included the InputHandler class, which was required for catching user input. Because of this, it was only possible to turn off all user input on the map, or use the actions of the default InputHandler. These default actions included sending Qt signals with data of the input of the user, and opening a pop-up on the map. These default actions were not very clearly documented.

The Qt signals sent from the InputHandler can be caught using a slot in the Python code. An example of a signal sent by the InputHandler is *mouseClickScreenPosition*. This signal sends the coordinates of an on-screen click within the map. If we were able to connect a slot to this signal it would be possible to add images or icons on the map and make them selectable. This signal could not be reached as no reference to the InputHandler was available in the bindings.

The MarbleWidget class, which is accessible and extendable within the Python code, contains the signal *mouseClickGeoPosition* providing the geographic location on the map on which a user clicks. *mouseClick-GeoPosition* is a signal sent from the InputHandler and resent by the MarbleWidget This signal can be used to connect clicks on the map to locations the user clicked on The slots connected to the *mouseClickGeoPosition* were never called and debugging showed that the signal was never sent.

This problem was solved by making use of the C++ source code of the MarbleWidget. A function within the MarbleWidget revealed a function with a very specific string required as argument before it would send the signal. This string was the signature of the *mouseMoveGeoPosition*, which sends all movement of the mouse above the map.

The default InputHandler also raises calls to other functions that change the view of the Marble widget. This could for example cause a pop-up with information about a location to show if the user clicked on an important part of the map. Since this pop-up restricted the view on the map, especially on smaller screens, this function needed to be removed. This was done by not allowing the PopupLayer in Marble to render by removing these lines from the Marble source code. Changing the Marble source code is not a durable solution, though for the functionality to be available until it is possible to do this through the Python code, it was required to do so.

PLACEMARKLAYER AND GEOMETRYLAYER

Marble contains a layer interface, allowing to create overlays over the displayed map. Some types of layers have been implemented already, including layers to show the terrain of the map, pop-ups or placemarks. In the plugin it was planned to use a placemark to show POI on the map. The bindings for the placemark layer were not included in the bindings, rendering it impossible to create this layer within Python.

This was fixed by creating a POI layer which subclasses the standard empty layer of Marble. Within this layer a reference was created to the Marble model. This model includes "documents" containing information about the features that will need to be rendered on the map. By adding the placemark to this model and toggling it's visibility on the map within the placemark object it is possible to use the placemark feature without using the default placemark layer.

5.3.2. DATABASE

The TRADR consortium uses two databases to save the data gathered by the robots and humans. These databases are a low-level[23] database and a high-level[27] database. These two databases should have a single point of access from the TDS, which was planned to be done by using a Model-API. This Model-API was supposed to be created by a member of the TRADR consortium. As the development phase progressed, it became clear that this API would not be finished on time to use in the TDS. Therefore direct queries to the databases were required to make use of them. To make a connection to the databases, it became apparent that previously created code by the TRADR consortium could be used.

A visual representation of the data flow between the databases and the TDS is shown in figure 5.8. Between the databases a script is running, which creates a 'bridge' between the high-level and the low-level database. This bridge keeps the information from both databases synchronized. As soon as new information is written to the low-level database, meta-data is extracted and entered into the high-level database through the bridge script. Also a reference to the newly added item is inserted into the high-level database, to connect the meta-data to an actual object in the database.

The code for the connection for the databases is saved in the DbQueries class. This class calls to other classes which establish the connections and starts polling the high-level database for new information. If new information is available or a query has been done manually, the high-level database returns the ID and requested meta-data to the DbQueries class. Using this ID, the actual data can be retrieved from the low-level database.

The DbQueries class also contains functions allowing to query the high-level database, without too much redundant code. This has been done by creating a general set of queries such as getting information and updating information. These queries receive a type as parameter, allowing a single function to query all types of data.

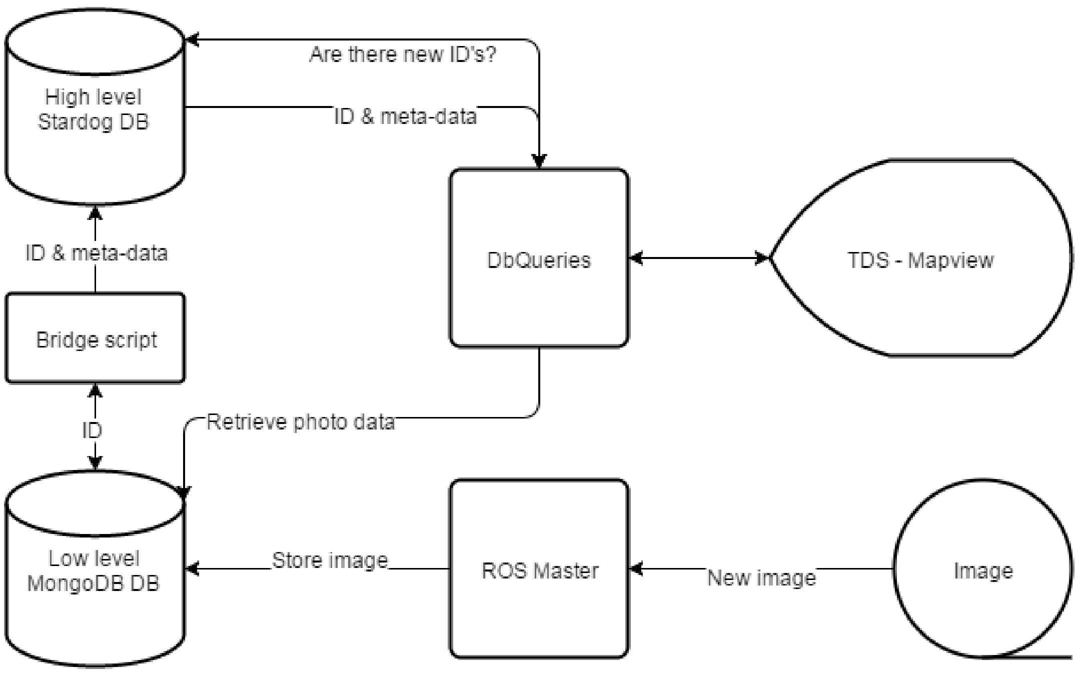


Figure 5.8: The flow of data between the high-level database, low-level database and the TDS

The data returned by a query was formatted in Extensible Markup Language (XML). To interpret this data, a Python module called Untangle has been used. This module parses the XML to a Python object, allowing easier navigation through the structure. It was chosen to parse it like this since the structure of the XML always has the same format for all queries, therefore allowing for static navigation through the returned XML string.

This flow of data had already been decided upon before the research phase, though this was expected to be handled by the ModelAPI. As the ModelAPI was not available during the development and since the bridge script and classes for the database connection were already available, it was chosen to create the flow in this manner. Also, since the ModelAPI is still planned to be created later on, the flow should not differ too much from the flow the ModelAPI will be using. This would possibly create difficulties when adapting the TDS to the ModelAPI.

Due to the model API not being finished and the previous code not being completely stable within the TDS, it has been decided with the client that the connection to the database will be finished after this project. This to make sure a fully stable and functional version will be delivered to the TRADR consortium.

6
Concluding Phase

The final phase of the project is the concluding phase in which testing is finished, the feedback from SIG is processed and the product and project is finalized. This chapter opens with a report on performed testing in section 6.1. After this the feedback from the first and second SIG deadline is given, including the actions for processing this feedback. This can be found in section 6.2.

6.1. Testing

After a product has been made, the client has to be convinced that the product does what it is expected to do. The modular independence created in the code of the product should make testing easier, as high coupling possibly introduces fault-proneness.[28]

Three kinds of testing are performed:

- unit testing
- integration testing
- acceptance testing

The most basic form of testing is unit testing. This kind of testing tests individual units of source code to determine if they are fit for use. An example of this is a very basic function which sets a variable with the given parameter. To test this function it is executed with a known parameter followed by a check if the variable now actually has the same value as the parameter.

Most test cases in the product are unit tests. This created some problems in the beginning as the production application initializes every bit of the program before using it. This was not the case in the test suite though as this is run without all the initializations to test only the part that requires testing. Also many libraries could not be imported which let all tests fail. The solution was to manually import the right libraries again and to make a setup for all tests, which initialized only the parts needed for that specific test.

A more extensive form of testing is integration testing. This kind of testing combines several software modules and test it as a group. While individual methods can pass a unit test by themselves it is possible that they provide a wrong result when combined with other methods. Implementing many integration tests is a quick and efficient way to achieve high test coverage and to make sure multiple modules communicate correctly. In this project not much integration testing has been performed. This was planned in to be done in the last week, but there was not enough time left due to the sudden database integration. This is further described in the next section, section 6.2

The last used form of testing is acceptance testing. The goal of this testing method is to determine if the requirements are met. This is often executed before delivering the product to the client. While there is software to test for example GUI's this was not done because of the lack of time. Instead the team decided to perform manual testing on the product which revealed several small issues, though these were mostly graphical issues which could be solved quickly.

After the project has been finished, an integration meeting will be done in which all systems created by the TRADR consortium will be tested. The TDS will be tested here too. These tests in a simulated environment will make it easy to double check acceptance tests. During this meeting or a following one, the product will be tested by the end users and more feedback will be received after this.

The tests resulted in a program that has as few bugs as possible. Since the amount of tests did not cover the full code, it is still possible that the code contains bugs that have not been found. The testing results will be further elaborated in section 6.2.

6.2. SIG FEEDBACK

During the project two submissions were made to SIG. SIG has designed the SIG 5-star classification system for software quality, based on the ISO/IEC 25010 model. Among other things, they perform static code analysis. Submissions have been made at the 75% mark and at the end of developing. The SIG feedback from the 75% mark is as follows (in Dutch):

"De code van het systeem scoort vier sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is.

Als wij naar de deelscores kijken, zien we dat ze allemaal bovengemiddeld scoren. Dit betekent dat jullie de belangrijkste onderhoudbaarheidsprincipes nauwkeurig hebben gevolgd, complementen daarvoor.

Het is ook goed om te zien dat jullie testcode hebben geschreven. Jullie zitten nog wel wat onder de optimale verhouding testcode/productiecode van 1:1. Voor de evaluatie van de tweede upload zou je nog eens kunnen kijken of de huidige testcode al de belangrijkste functionaliteit van de code raakt. Het toevoegen van testcode is over het algemeen altijd aan te raden, want meer tests leiden tot minder fouten in de code.

Over het algemeen scoort de code dus (ver) bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase."

Translated in English:

"The code of the system scores four stars (authors: out of five) in our maintainability model, which means that the code is above average maintainable.

If we look at the sub scores we see that they all score above average. This means that you accurately followed the most important maintainability principles, my complements for that.

It's also good to see that you have written test code. You are however below the optimal 1:1 testcode/productioncode ratio. For the evaluation of the second upload you could look if the current test code hits all the important functionality of the code. Adding test code is in general always advisable, as more test lead to less errors in the code.

So in general the code scores above average, let's hope you can maintain this level during the rest of the development phase."

The result of the first SIG evaluation was expected, as we hurried to write tests before the first submission. As the rest of the code scored well, we tried to maintain every coding pattern we had. In the last week of the project, the planned time to write the majority of test code, it became clear that we would not receive a database API as was discussed in the beginning of the project. Next to finalizing the program and writing the database code there was little time to also finish writing test code. As most program functionality was coupled to the GUI it was decided to manually test the code with the help of the written use cases.

Because of a lack of time there was actually less coverage in the last submission than in the first one. More tests are present, and more code is being tested than in the first submission, but also more production code was added in the meantime which results in a lower coverage percentage. At the first SIG deadline a code test coverage of 65% was achieved. At the second deadline, the tests resulted in a coverage of 55%.

The second SIG feedback was as follows:

"In de tweede upload zien we dat de omvang van het systeem gestegen is. Het systeem scoort nog steeds vier sterren, wat betekend dat het bovengemiddeld onderhoudbaar is.

Er is echter een stuk daling in de score op te merken. Deze is met name toe te schrijven aan Unit Complexiteit. In de bestand database.py bijvoorbeeld is er een unit 'run(self)' die bovengemiddeld complex is. Het opsplitsen van dit soort units in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt.

Wat betreft de testcode, is het goed om te zien dat er een stijging in het volume van de testcode is.

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie zijn meegenomen in het ontwikkeltraject."

Translated in English:

" In the second upload we see that the size of the system has risen. The system still scores four stars which means that it is above average maintainable.

There is however a noticeable drop in the score. This is mainly due to Unit complexity. In the file database.py for example there is a unit 'run(self)' which is above average complex. Splitting this kind of units in smaller pieces ensures that every part is easy understandable, testable and therefore easier maintainable.

With regard to the test code it is good to see there is a rise in the volume of the testcode.

From these observations we can conclude that the previous recommendations from the last evaluation are taken into account in the developing process. "

The team agrees with this code evaluation as the database code can be improved. This is mainly because this code is not completely written by the team, but an adjusted version of code used in other TRADR software. This is however not a problem as this will be replaced later by the Model-API.

7 CONCLUSION

This chapter gives a conclusion to the research project. This will start in section 7.1 followed by what we learned from this project in section 7.2 . The chapter ends with section 7.3 where is described which knowledge from the bachelor program was used.

7.1. CONCLUSION

In the beginning of this report the following research question was formulated:

How to create a modular GUI for use in a robot-human-team-based Search and Rescue operation?

Answering this question began with defining all requirements which defines the goals of the project. The team managed to implement all requirements from the 'must have' section (see subsection 4.2.2) and many others. This is one of the success criteria stated in the definition of done in subsection 4.2.3. The requirement for the database will be finished after the project as mentioned in the last paragraph of subsection 5.3.2.

The second criterion in the definition of done is passing all acceptance tests. Although testing was done less thoroughly than planned (see section 6.1), we managed to pass manual acceptance testing based on the use cases. These use cases were made based on the requirements in section 4.2.2. As the team held weekly meetings with the client discussing the progress, the implementation of the requirements has also been verified here.

The third and fourth criteria are also met as the application runs on the specified hardware and does not have any outstanding high priority, severe or program breaking bugs. A side note is that the database functionality still has some issues. This is further discussed in the Discussion in section 8.1.

After the research question was formulated sub questions were made. The first sub question is how to create a modular GUI. As stated in section 5.1 this can be done by building each part of the program as a plugin using the Model-View-Presenter architectural pattern and a general 'add-plugin' function. This function makes sure that each part is loaded correctly and separately in the right order. By building with plugins that can work independently, modularity is implemented. Also the usage of RQT contributed to an modular user interface.

The second sub question is how to display the information needed by these rescuers in an effective way. The answer to this question, given in subsection 4.3, is that non-invasive messages should be used, that the map overview should be the center of the application and that it should be possible to minimize distractions. This can be done by allowing the user to view only what is needed at that time by minimizing information views and/or toolbars which are unnecessary in a specific situation.

The last research sub question; *"How to integrate this system into existing systems TRADR already uses?"*, is answered throughout the report. To use the TDS alongside other user interfaces such as the OCU, the TDS has been built as an RQt plugin, allowing the user to open multiple interfaces in a single main window. To integrate the TDS in the already existing TRADR architecture displayed in figure 4.1, a connection to the databases has been made as described in 5.3.2, allowing the TDS to receive all information stored, including saved images and points of interest.

The research questions have been answered and the criteria from the "Definition of Done" are met, so now can be concluded that the problem has been solved, and thus the research question has been answered. The client, the TRADR consortium, now has a modular base to build on top of and to use the coming years during their project, which was the goal.

By making this product and implementing this as part of a larger system it is achieved that Search and Rescue missions can be executed more efficiently and thus potentially lives can be saved.

7.2. LEARNINGS

By doing this project we learned much from both a Computer Science perspective as project experiences perspective. In the research phase we experienced how to work together with other international organizations. This was for example done by working at TNO and going to Fraunhofer in Germany. By doing this we also learned to work together with various kinds of people; researchers, students and people from business environments.

We also learned to respect deadlines even more than before, as failing to meet these deadline caused many problems during the project. We learned during the development phase that we should communicate earlier and more clearly if there is a problem. We also learned not to promise to create something without being sure it is even possible. Next to this the experience with programming languages (mainly python) and applying architectural patterns has increased a lot as we worked through all functionality.

7.3. USED KNOWLEDGE

As this project is the last course of the Computer science course previously accumulated knowledge had to be used. Next to all project experiences, the Software Quality and Testing course provided knowledge of using Git, different types of testing and how to test in general. We used knowledge from Software Engineering methods about benefits/drawbacks of certain design patterns, as also the use of Scrum.

8 Discussion and Recommendations

To conclude the report the product itself will be discussed in section 8.1. This is followed by recommendations for future development in section 8.2. To finalize a reflection on the process is given in section 8.3.

8.1. Discussion on product

In the finished product, some bugs have been discovered already. One of these bugs takes place when using restricted functions with the info rights management. When information is retrieved from the database, it is possible that information is not shown in the TDS. This is due to the fact that the retrieved information is added to the program using the same functions as the functions the user can use to add data manually to the TDS. If this function is restricted, the function will not be executed by the program, therefore the received data will not be added locally and shown. This should be fixed before the info rights management is actually going to be used and it can possibly be fixed by overriding the permissions from the user if the function is called by the database if the user is allowed to receive the information from the database.

Another bug, which is of importance, is the fact that the program does not start without a database connection. The database connection is created using already available code from the TRADR consortium. This code ends up in a waiting loop if the database connection is not found. Because of this, the initialization of the TDS is halted until a database connection is established. Multiple possible solutions have already been tried, such as using threading to separate the main thread and the database connection thread. Using threads caused synchronization issues and in some cases the database connection would not be available to the code. Since the connection between the database and the TDS was created in the last moments of the development, we were not able to fix this bug before delivering the product. It could possibly be solved using threads in some other manner or on a different part in the code, specifically the database connection code from the TRADR consortium. This problem could also be fixed if the API is written for communicating with the database from the TDS. The blocking loop when querying the database when no connection is established and when establishing the connection should be taken into account when writing this API.

The program also is unstable without a database connection. As most of the functionality of the program is using the databases as ground truth, a lost database connection can cause the program to receive invalid data or block the main thread. This can cause the program to throw exceptions. Most of these exceptions are caught properly and shown in the TDS, allowing the program to proceed, though in some cases the program can still crash. This problem can be solved when the API for the databases returns a proper value to indicate if something went wrong.

8.2. Recommendations

During the project, an API would be provided to query the database. This API was created to relieve the developers from having to query the multiple databases manually. It would be used as main access point to the database, providing in all required functionality.

As this API was not created yet at the end of the development of the TDS, direct queries to the database have been created to make the TDS functional. These direct queries do make the source code of the TDS more difficult to read, as well as making it harder to change the database or database structure. Therefore we recommend to still create an API for the connection with the databases. Not only does it allow for easier maintenance than the currently created database module, it also makes extending or adjusting the TDS easier.

Many problems also became apparent when working with the PyKDE bindings for Marble. Problems appeared when trying to access the input handler or placemark layers. The person maintaining the TDS should keep track of the development of the Marble and the PyKDE bindings. New and updated bindings may solve many bugs which occurred due to missing or incorrect bindings. These bindings can create easier and more efficient solutions to problems, like adding or editing placemark object.

To make the GUI more modular, all individual toolbars could be added to RQt separately. By doing this, every single plugin can be added when required. Due to RQt's modular interface, all these toolbars can be relocated and resized independently of the TDS itself. By doing this the OCU can for example be added between the toolbars and the map. That RQt is able to do this without breaking references between different modules was not known to us during the development phase. Therefore we did not have time to change the code in such a way to accommodate this. It can be done by replacing the MapViewPlugin class with multiple separate classes for each toolbar or function that requires a view and adding them separately to RQt.

8.3. Reflection on process

Overall we think the process was satisfying. We had a good relation with the team members of the TRADR consortium which resulted in friendly informal and thereby efficient conversations. Something we really appreciated doing is writing down everything that was said as our memory is quite bad. This way we made sure that everything that was discussed was also actually processed. Also appointments were clear for both parties by doing this.

Next to this, we were very proactive in seeking help when needed, though this could have been better in the first weeks when a lot of software was not working. In the last few weeks we discovered that we did not presented our problems clear enough as our coach didn't realize that essential functionality was not working at the time.

We are very enthusiastic about the collaboration between us team members. As we are only with two people decision could be made quickly. We worked together every day and had fun while doing that.

An aspect that could be improved in the process is the planning of time. We needed more time in the research and concluding phase than expected which didn't leave enough time for the development phase as this was shortened by two weeks. This meant that the last two weeks were done in a hurry which resulted in a lower quality of coding and writing than can be achieved.

Another point of improvement is knowing when to stop trying if something is not working. We spend a lot of time on certain functionality while in hindsight this could not have worked or another solution would be better. We also did this on functionality which is not in the must haves section of the requirements, while more important functionality was not implemented yet. This time could have been better spend.

A
Project description

The TRADR project is a European collaboration between 8 different universities on the topic of Search & Rescue robots. At TU Delft we are developing an agent-based system to enable robots (ground and air) to operate in a team of human firefighters during Search & Rescue missions, e.g. after a fire or earthquake. The system consists of a multi-agent layer for teamwork coordination, an RDF database for ontology modeling and an adaptive user interface (UI) that supports coordination of teamwork between humans and robots, provides tactical information and can be operated by firefighters in the field (e.g. through tablet/smartphone).

The problem that needs to be solved concerns the question of how to design a system that enables firefighters to work effectively with robots during Search & Rescue missions. Concretely, the BSc project consists of the design, implementation and testing of a back-end system and prototype UI which combines data from the agent-layer and RDF database, and presents this to users on a tactical information display.

The agent layer coordinates teamwork for both people and robots, and individual agents decide what to show to specific users (when and how, eg. draw attention vs subtle). The RDF database allows for model storage and reasoning and connects to the UI.

The UI is envisioned as something like a GIS map with multiple layers of information (e.g. team-members, infrastructure, water points etc), that can display a variety of information, depending on the user and on the context, and that can be used to coordinate teamwork. Based on experience from earlier tests, there are a number of design requirements, such as:

- connection to RDF database
- map (eg. Google maps) as a base-layer
- multiple layers showing different kinds of information
- widget-like setup to easily switch on/off different components
- the Marble framework will be used as a basis
- coding can be done in C++, Qt Quick (QML) or Python

source: Bepsys: http://bepsys.herokuapp.com/projects/view/102

B
Scrum sprints

B.1. General planning

WP1: 6 Juli

- map tools 4h
- shortcuts to locations 3h
- Communication with other Ros Nodes 4h
- a structure allowing plugins to extend the program 1h
- a layered structure 6h
- Modular set-up of interface 2h

WP2: 27 Juli

- an indicator of status of actors
- Icon features
- display of photos
- Info rights management
- faulty connectivity/power notifications
- Database: information properties

WP3: 3 Augustus

- handheld device support
- setup guide
- multiple instances

B.2. Actual planning

These are the results after the scrum meeting at the beginning of each week

B.2.1. SPRINT PLAN WP1

Bucket until WP1

- a layered structure
- Maptools
- shortcuts to locations
- a structure allowing plugins to extend the program
- Modular set-up of interface
- Communication with other Ros Nodes

WP1 Plan

- a layered structure
- Create single layer
- Create generator
- Layer selection
- Maptools
- Measure Distance
- Designate Area
- Scale
- shortcuts to locations
- a structure allowing plugins to extend the program
- Modular set-up of interface

Comments: Many of the first waypoint functionalities were already available after installing all existing plugins.

WP1 After recap Due to various non-existent functions which should be available normally, and way too much research time needed to get layers properly working, time approximation was way off. Programming speed is going up gradually and approximations become better.

We weren't able to finish:

- Layered structure -> Main functionality is now available, but more work is needed to have it working correctly
- Maptools -> Many of the build-in functions do not work as expected or not at all. Need to rewrite these functions completely

B.2.2. SPRINT PLAN WP2

WP2 - Week 1 Plan

- Notification- & Errorcenter 4h – Marnix
- Layer structure 8h – Jamey
- Create POI layer 6h – Jamey
- Layer z-index switcher 4h –Jamey
- Database 12h – Jamey & Marnix
- Go-to from notification 1h – Marnix
- Create restricted layer 0.5h – Jamey
- Connect to database - Shortcuts to location 2h – Marnix
- Highlight POI layer points in layer structure 8h – Jamey
- Measure distance rewrite 5h – Marnix
- Scale bar 2h – Marnix

WP2 - Week 1 Recap

Notificationcenter now only needs a database connection. Layer structure got fixed after being plagued by a harsh bug. Database is thought trough but needs to be talked through first with Timea. Documentation got added but needs to be worked out more.

Not (completely) finished:

- Layer z-index switcher
- GO-to from notification (button exists)
- Database
- Highlight POI layer points in layer structure
- Measure distance rewrite
- Scale bar

WP2 - Week 2 Plan

- Database 9hr – Marnix
- Layer z-index switcher 5hr – Marnix & Jamey
- POI Layer - add POIs 6hr – Jamey
- Go-to from notification 1hr – Marnix
- Maptools 1hr – Marnix
- Communication with ROS 4hr – Jamey
- Tracking POI Layer 3hr – Jamey
- Faulty-conn. Notification –2hr – Marnix
- Displaying photo's 10hr
- Actor-information 9hr – Marnix & Jamey

Plan for Monday: Finish documentation as far as possible WP2 - Week 2 Recap Beginning of the following

week the SIG deadline is planned. Therefore a lot of refactoring has been done at the end of the week, instead of much new functionality. We're still waiting on the API for the database, therefore some functionality cant be finalized/tested. Though the DB is already installed. The structure has been discussed already. Communication with ROS has been moved to a lower priority. Displaying photo's became top priority for next week now. Layer z-index switcher is impossible due to Marble's placemark rendering.

WP2 - Week 3 Plan

- Displaying photo's – 15 hr
- Database – 5hr (Awaiting Model API)
- Actor-information – 9hr
- Tracking POI layer – 5hr
- Allow minimalisation/re-opening of dock-widgets – 3hr
- Faulty-conn. Notification – 2hr (Awaiting Model API)

Remark: Less hours are assigned to this week, due to the need for more time for writing tests and documentation.

WP2 - Week 3 Recap
The first 2 days of the week were spent on writings tests before the first SIG deadline. This caused some delays in the planning. Displaying photo's has gotten way more difficult than expected due to unavailable bindings for the inputHandler and PlacemarkInfoDialog. A workaround is in the pipeline. Database Model API won't be implemented before the end of this project (because of no good implementation of lazy evalualtion in Python), therefore we're going to try to directly communicate with the DB.

B.2.3. SPRINT PLAN WP3

WP3 - Week 1 Plan

- Displaying photo's – 13 hr (including trying to make the placemarks clickable instead of list of placemarks)
- Database – 10 hr (Including connectivity notification)(Model API cannot be used, manual stuff)
- Actor-information – 2hr
- Tracking POI layer – 4 hr
- Allow minimalisation/re-opening of dock-widgets – 3hr
- Process SIG feedback – 6hr

Although the following was not really discussed in a scrum meeting anymore, the following was done in the last week(s):

- Database implementation
- Login screen
- Connectivity notification for database
- Process SIG feedback
- finalized documentation and made read me file. Removed unused code.

C

Appendix C: Research Report

CONTENTS

1 INTRODUCTION

1.1. PREFACE

This document is the research report of the TRADR assignment as part of the Bachelor Thesis course at the Delft University of Technology, The Netherlands. This course is the last part of the Bachelor's programme, allowing the students to prove what they learned and are worthy of receiving the Bachelor's degree.

This report provides an introduction into the problem we are going to solve, including how it is going to be done and why it will be done like that. The goal is to give an insight into what has been researched to find a solution and to show the chosen way is the best way to do it.

1.2. INTRODUCTION

Search and Rescue operations can be very dangerous. The in-field rescuers can be exposed to fire, hazardous materials, unstable structures and other hazards which cannot be seen from their field of view. To make the in-field rescuers' job less dangerous, robots are being used more often to do the possibly dangerous or impossible jobs instead of the humans. The robots do not just make the job less dangerous, it might also speed up the mission, which could save lives.

The Long-Term Human-Robot Teaming for Robot Assisted Disaster Response (TRADR) project is an European collaboration consisting of 12 partners from 6 countries on the topic of Search & Rescue robots.[1] Among them are 5 universities, 3 research institutes, one industry partner and three end-user organizations, representatives of fire-brigades from Germany, Italy and the Netherlands. The project started on November 1, 2013, which means that at the time of writing this report the project is in its second year, out of four. TRADR is developing an agent-based system to enable robots (ground and air) to operate in a team of human firefighters during Search & Rescue missions. It builds on the research and experience of the NIFTi project which focuses on human-robot teaming.[2]

A use case of TRADR is responding to a medium to large scale industrial accident, which can stretch over several days in dynamic environments. Various kinds of robots collaborate with human team members to explore the environment and gather physical samples. Other use cases include man-made or natural disasters such as earthquakes, floodings or major traffic accidents.

1.3. PROBLEM DEFINITION

An important part of the TRADR project is creating an overview of the mission area and communicating important information in an effective way. To do this, a modular Graphical User Interface (GUI) for the TRADR system is desired on which a map of the area can be displayed, along with various other information such as locations of casualties, danger zones, photos of disaster sites etc. This information should be easily accessible by the in-field rescuers and new information should be easy to insert and send to all other emergency responders involved in the mission.

This GUI for the TRADR-project will be called the TRADR Display System (TDS) and will consist of two major parts; The MapView and the Operator Control Unit (OCU). The OCU has already been created. This bachelor project is aimed at creating the MapView plugin to run alongside the OCU. This plugin will be required to be extendable, therefore it will need a modular design. It also will need to be a sturdy and easy to use system, which requires no deep technical knowledge. The main question of the project therefore is:

How to create a modular GUI for use in a robot-human-team-based Search and Rescue operation?

1.4. BACKGROUND

In the years before the TRADR project was the NiFTi project.[3,4] The goal of the project was to bring the human factor into cognitive architectures, while developing robots capable of collaborating with humans in teams. The TRADR project builds onto the experiences gathered in the NiFTi project. They are both aimed at using human-robot cooperation in Urban Search & Rescue missions.

One of the conclusions of the NIFTi project was that information persistence is very important, as the missions can have a duration of longer than a day. Without this, much information can be lost or forgotten as the time goes by. The TRADR project aims at adding persistent models to the experience gathered in the NIFTi project to create the ideal human-robot cooperation in search and rescue missions.

2 REQUIREMENTS

This section defines the use and requirements of the application. In subsection 2.1 the actors of the system will be described. After that all requirements will be listed in section 2.2. Next the driving design principles will be discussed in section 2.3 and lastly, a general overview of the system design is given in section 2.4.

2.1. ACTORS

The actors of the system are different per location, as the system is used by fire brigades of different countries. In general, the following actors will be present:

1. Mission commander: Gives orders to team leader and has the overview. The commander sees all the data a technical team (who selects useful images/information from incoming data streams) has prepared for him.

2. Team leader: Connection between mission commander and team (which are the robot operators and data pre-selection persons). The team leader does not access dangerous areas.

3. In field rescuer: Retrieves mission information from the TDS for in field use. It is possible that in field rescuers have certain specialties and capabilities. These therefore might use the TDS in a different way.

2.2. REQUIREMENTS

This section lists the requirements for the product in subsection 2.2.1 and describes the criteria to label the product as finished in subsection 2.2.2. Use cases are made based on these requirements which can be found in Appendix A.

2.2.1. MAIN REQUIREMENTS

The requirements for the product are categorized through the MoSCoW principle:

MUST HAVES

The product must have:

1. a modular set-up of interface: be able to show or hide certain components and personalize interface according to role, personal preference and device
2. a layered structure (where layers can be switched on or off) with
   - base map
   - location of team members
   - info icons (Points of interest such as victim, fire, hazardous material, obstacle, remark, photo)
3. info rights management (who can see what, based on different roles)
4. give a notification in case of faulty connectivity and reestablishes it when available.

5. display of photos
6. a structure allowing plugins to extend the program
7. the possibility to run multiple instances on different machines, while displaying the same information on the multiple instances.
8. a setup guide.
9. a communication ability with other ROS nodes (e.g. to display a videostream)
10. handheld device support: touch control and keyboard on screen when using a touch screen
11. user friendly feedback when an error occurs.

Also, the product must:

- run on Linux
- use KDE Marble
- be implemented as a set of RQT plugins
- use databases as 'ground truth' for data retrieval and storage, in particular the Stardog (RDF) and low level database from the existing system.
- be fully documented using Sphinx and Rosdoc

SHOULD HAVES

The product should have:

1. the following added layers:
   - layer for photos, including clustering where multiple pictures have been taken (like in Google Maps)
   - layer for robot trajectory traces
   - area indication ("working agreements") (go, no-go, unstable, searched, robot autonomy)
   - user input on the map (Text, notes, drawing, photos)
2. the following icon features:
   - icons should be informative (e.g. thumbnail icons from photos, indication of robot control)
   - icons should scale naturally
   - possibility to lock icons (as to not move them by accident, possibly default)
   - show field of view/orientation for photos and robots when data is available
3. the following properties for information in the database:
   - timestamp of entry
   - name of actor who entered the information
   - visible in TDS
4. an indicator for reachability of actors
5. shortcuts to locations
6. map tools (scale info, distance measure)
7. GPS-coordinates visible within TDS (when adding icons and current view)

COULD HAVES

The product could have:

1. Robot status info screen (battery, connection status etc.)
2. time line of events (possibly within main TDS for team leader)
3. possibility for situation report or briefing
4. representation of team management information (e.g. showing team members; show goals, tasks and status)
5. global search function (e.g. search for data, text, objects)
6. Streetview-like option to view specific area as 3D map (Overlay the 3D point-cloud map with the 2D plan)
7. information from external sources (e.g. weather information)
8. possibility to show videos and play back videos
9. speech input (in cooperation with DFKI)
10. display of (co-)variance of GPS location
11. comment functionality on photos and other entities
12. simple image editor
13. multitouch functionality (gestures, pinch-to-zoom)
14. recovery options (e.g. using the last position of a robot when the connection is lost or power supply is not functioning)

HARDWARE

The hardware on which the application must run is:

1. A computer system running linux 14.04 with wifi.
2. A tablet running linux with wifi.

2.2.2. DEFINITION OF DONE

This subsection will state the criteria which the finished product has to meet to be called done.
For the product to be finished, the system has to:

- have all *must have* requirements from subsection 2.2.1 implemented
- pass all acceptance tests
- have no high priority, severe or program breaking bugs left outstanding.
- run on the required hardware as stated in subsection 2.2.1

2.3. DESIGN GOALS

During development certain design goals are taken into account. These goals are high level and can be thought of as the driving principles during the project. This is to ensure a stable and fast performing program which is pleasant to work with for end users. The following subsections describe these goals. Security is not an important design goal as the client stated that this is not a necessity for now.

2.3.1. MAINTAINABILITY

A very important design goal is modularity. A modular build application is a part of the 'must haves' in the requirements, so this will mean a lot to the client. Some advantages of a modular build application are that it can be easily expanded. Also, when something needs to be altered, it is possible to change only that specific part without the rest of the program being affected. This will be very useful for the client as expansion is planned right after we are done with the project. Modular independence also makes testing easier, as high coupling possibly introduces fault-proneness.[5]

To ensure that expanding and changing the application is an easy task, the code will be extensively documented in the form of comments, installation and setup guides and documentation in the version management system the TRADR-project team already uses.

The last design goal for proper maintainability is to ensure high code quality. This can be done by adhering to coding standards,[6] processing feedback from Software Improvement Group (SIG), doing static code analysis in the IDE and peer reviewing.

2.3.2. USABILITY

The application is going to be used in various and sometimes chaotic environments. The goal is to create an overview of the situation and thus the program should also be easy to use and to get information from. To do this, the meaning and function of used icons should be instantly clear.

To continuously keep an overview of the situation, users should not be disturbed by dialogs and should not leave the current context when trying to enter new information.

Adding to this, when an error is made by the user or system, a user friendly error-message should appear which is clear and noticeable, but not invasive.

2.3.3. PERFORMANCE

During a search and rescue mission all systems have to perform when needed, as failing hardware or software could mean loss of lives. This means the application has to keep functioning correctly even when things go wrong with external factors as for example a lost connection to the database. After such a failure, the system still has to display at least the known old information, and recover new information when the connection is reestablished.

When multiple instances of the TDS are active, the system will have to take care of consistency between them. By doing this, we make sure that everybody has the same information at hand. To do this, continuous updates have to be done. This cannot be done when a failure happens, so the correct and up-to-date information will have to be shown again when the connection is reestablished.

2.4. DESIGN OVERVIEW

As described in subsection 3.2.2, the system will be built as a set of plugins. This can be seen in a schematic high level overview of the system design in Figure 2.1. The list of possible icons, connected to the IconProvider plugin, is not exhaustive. The system will be built modular to allow easy modification of possible entities. The OCU plugin in the RQT package is already build, but as is works the same as other plugins and it is a part of the TDS, it is also shown in the overview for extra clarity. The IconProvider and the ToolProvider plugins are not in the RQT package as they can work independent of the MapView, OCU and NotificationCenter.

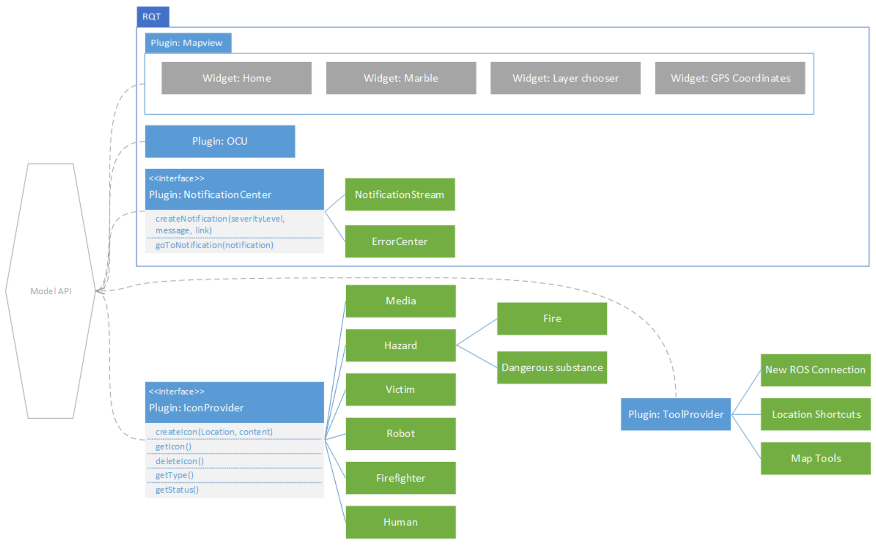


Figure 2.1: High-Level System Design Diagram

The system also makes connections to two databases. A low level database is mainly used for storing pictures and other sensor data, while the high level Resource Description Framework (RDF) database is used for storing references to the images, entities and their meta-data. The high level database is based on an ontology, so this can be used for reasoning. The info in the databases can also be queried from within the TDS. The communication to the databases will not be done by the TDS itself but through the Model API. This is indicated in Figure 2.1 by the gray dotted lines to the Model API. The Model API is not part of this project.

2.5. GUI DESIGN

The TDS should have a GUI that does not distract in any way from the task the user is performing. Users should be able to see the requested information when they need it and not be bothered by unnecessary messages or screen-covering notifications. Harrison tested how the Stroop Effect could be used in a GUI by using transparency.[7] With the optimal level of transparency, the optimal division between divided and focused attention can be achieved.

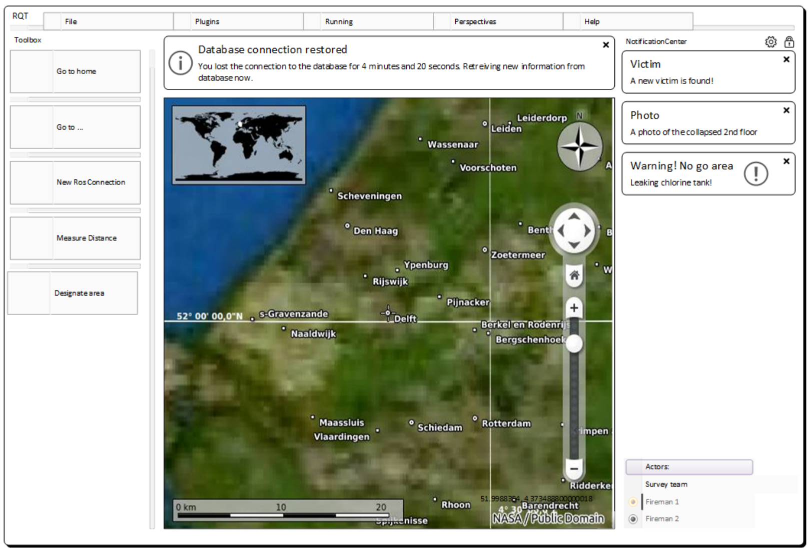
In the TDS' GUI, the focus should lay on undivided attention, as it can be hard to keep organized in the chaos of a mission. Blocking the view on a small bit of information can cause the user to miss important events. Therefore an interface with no screen-blocking notifications or pop-ups is required. Notifications will be given through a notification center, which does not create a pop up, but is displayed as continuously updating stream. This allows the user to see the notifications when he has time for them, and the system tries to make sure he does not miss any essential messages by showing how urgent a notification or message is.

Divided attention should be taken into account too. For example, the users which control the robots might need to use the TDS too, which requires them to use the OCU and the TDS at the same time. Splitting and arranging the windows will be done by RQT itself, as it contains a tiled window manager. This ensures the user can change the amount of screen he wants to allocate for the separate views. An option to lock and unlock the window size properties will be present to make sure that windows don't move accidentally when not wanted to.

Tools and extra information not directly linked to a location will be shown using (tool)bars. These might be slightly transparent if they lay over the map due to window sizes, though it should not be too hard to see which button is which tool.

Icons will create a pop-up if they are selected. This screen will show all information that is linked to that point in a non-transparent overlay. The user should be able to read the information clearly and efficiently, which transparency might counteract against.

In figure 2.2 the envisioned GUI can be seen. On the left is the toolbar for easy access to tools while on the right the notification stream is displayed. Also the actor reachability is displayed here. The most important information, the overview of the area, is in the center of the application. Above this an error notification is shown (when needed) in case something goes wrong within the system. The menu on top is an default in RQT.


Database connection restored
You lost the connection to the database for 4 minutes and 20 seconds. Retreiving new information from database now.

Victim
A new victim is found!

Photo
A photo of the collapsed 2nd floor

Warning! No go area
Leaking chlorine tank!

Actors:
Survey team
Fireman 1
Fireman 2

Figure 2.2: Envisioned GUI design

3
SOFTWARE

As the TRADR project is already in its second year, some decisions have already been made by the team about which software and frameworks to use. An analysis of the software follows in this chapter. In section 3.1 the software used for the project is described followed by the used frameworks in section 3.2. The chapter ends with section 3.3 in which a description of the languages in which the program is written is given.

3.1. SOFTWARE

MARBLE

Marble is an application that can display a view of the earth.[8] It can be used as an application of its own, but it also serves as an example of how to use Marble widgets in a self-made program. A big advantage of Marble is that is runs on every operating system that supports QT4, which is also used in the TRADR project. The reasons for using QT4 instead of the newer QT5 are described in subsection 3.3.1.

In order for Marble to work, some dependencies have to be installed. On a clean install of Ubuntu 14.04, the following packages are needed:

- SIP: A tool needed to create Python bindings for C and C++ libraries. In this case the PyQT library.
- PyQT: A set of Python v2 and v3 bindings for QT.
- PyKDE: Provides the python bindings for KDE, where Marble is a part of.

Together they combine all the advantages of KDE, Qt and Python.

3.2. FRAMEWORKS

A framework is a universal reusable software environment that provides certain functionality as part of a larger program. Two frameworks will be used for our development.

3.2.1. ROS

Robot Operating System (ROS) is a framework which is used to make it easier to program robots. It is being used for the TRADR project, and needs to communicate with the TDS. This is because ROS sends information between its processes by sending messages, also to and from the TDS.

In the current TRADR project, ROS Indigo is being used, which is not the newest version of ROS. Though, Indigo is the most stable version at the moment and recommended for stability.[9]

3.2.2. RQT

RQT is the base on which the program has to be build. It is required for the product as ROS is being used by the TRADR system as base.

RQT is a plugin for ROS which allows developers to build a GUI on the ROS platform in the form of plugins.[10] These plugins can be written in Python and C++, in combination with Qt. The plugins written for RQT can be used as a standalone program, or as a window docked inside of the RQT window. The TDS will consist of multiple separated parts; the OCU and the MapView. These two parts are likely to be used together at the same time, on the same machine. To make it easy to use these programs together at the same time, RQT allows arranging multiple plugins around the window by clicking and dragging. Due to the fact that this feature is already built in, the program will be build to be embedded in the RQT window.

3.3. LANGUAGES

The following two subsections will describe why the source code of the project will be written in Python and QT.

3.3.1. QT

Qt is a technology which makes it easier to create cross-platform programs with the same user interface. Since the TDS should work on Linux desktops, but also tablets and possibly even smartphones, Qt is a good choice for creating the GUI of TRADR as QT supports all these platforms. Qt is also required when using marble as it is built on it.

The newest version of Qt is Qt5. For this however, no proper bindings are available to use with Marble. Therefore, we are required to use Qt4, in combination with the PyQt4 bindings. Qt4 has a worse performance than Qt5 and lacks some new features, though these are not needed explicitly. The performance will not become a big problem as the TDS will not demand as many resources as popular Qt based programs. Therefore porting the Qt4 source code to Qt5 will not be done during the development of the TDS.

3.3.2. PYTHON

The main programming language of this project will be Python2.7. This is because RQT can be used with Python or C++. Python will require much less overhead code to create the same functionality. C++ libraries often can be used with Python too, using available bindings. As these are available for every required library the TDS will be dependent on, using Python should not form any problems.

There are two broadly used versions of Python available at this time; Python2.7 and Python3.4.[11] Python2.7 can be seen as legacy, though it is still widely used and will still be receiving bug-fixes and support. Python3 fixed many of the bugs and problems Python2.7 has, though these bugs do not have to be disruptive for creating the TDS

The preferred version of Python would be version 3, though the Python3 bindings of the KDE are not created for Marble yet. This forces us to rewrite the bindings of Marble, or to use Python2.7. The last option requires much less time and has a lower risk, as rewriting the bindings might be a very time consuming task which can turn out harder than expected.

For building the TDS, Python2.7 will be used in combination with the `__future__` import, which allows easier porting to Python3 as soon as the bindings are available.[12]

If the bindings are created long before the final deadline, the choice to use Python3 may still be made. This is however not probable.

4
Testing Frameworks

There are a couple of testing frameworks available to use on Python 3. These for example include Python's standard unit test module -better known as PyUnit-, Nose and Py.test. Each of them has their own advantages and disadvantages. A tool to test the GUI is also needed, since the main functionality of the system will be a GUI

4.1. PyUnit

The standard unit tests module (also known as PyUnit) is available in the python libraries. This module is based on the same framework as the Java testing framework JUnit. In comparison with other available modules for Python, unit test requires verbose code for many simple tasks. The main advantage is that it is included in the Python libraries and therefore doesn't need external libraries. The fact that it needs much verbose code to create simple tests and that it is solely capable of creating unit tests makes it not ideal.

4.2. Py.test

Py.test is one of the most used testing frameworks for Python nowadays. It allows many types of testing including unit tests, doctests and nose tests. The Py.test framework has been properly maintained and documented and therefore using it is easy to learn.

4.3. Nose

The Nose framework is a cloned version of Py.test and still holds to the same philosophy as Py.test. It is able to recognize and use PyUnit tests as well as Py.test tests and it is able to provide the same testing techniques as Py.test. The code needed for the tests is nearly identical between Py.tests and Nose, therefore there is not much verbose code needed for Nose either.

4.4. QTestLib

QTestLib is a library that can be used to test programs written in Qt and PyQt. It offers features to test GUIs, like performing mouse- and keyboard-events. This allows the GUI to be properly tested, which is a must for this project. QTestLib can be used in combination with PyUnit tests, but as these tests can be interpreted by Nose, it works in combination with Nose too.

The Nose framework in combination with the QTestLib will be used for this project. The Nose framework has been used in previous code of the existing TRADR project and offers all needed functionality to test Python code properly. To keep the usage of frameworks consistent, the choice for Nose seems to be the best one. QTestLib will be used to offer GUI-testing abilities to Nose.

5 Documentation

The TRADR project is a collaboration of multiple organizations. Since the amount of source code is growing, and the TDS will be further developed after we are finished with this project, proper documentation is critical. There is a wiki available for the project members, including documentation of the code and other information about the project or meetings. The documentation for this is made using automated tools to reduce the workload and to ensure consistency. Therefore we will make use of Sphinx (in combination with the *autodoc* plugin) and rosdoc-lite. Rosdoc will be needed to allow Sphinx to run on the ROS packages. The generated documentation will be placed in the RedMine wiki.

Another possibility was to use *Doxygen* instead of Sphinx. Doxygen is a documentation generator mainly used for `C++`, but it is also stable on multiple languages including Python. There are many plugins that allows RedMine to import the generated HTML documentation quickly. The choice not to use *Doxygen* was made as it is not native for Python, whereas Sphinx is. It should not create a problem if it would be used, but as the other documentation already available on the RedMine wiki has been created using Sphinx, this got the preference for consistency.

A

Use Cases

Coordinates visible within TDS:

Prerequisites:
A role is chosen
Map layer is active

Story:
The current gps coordinates of the center of the area are always visible at the down-right corner of the screen.

Map tools:

Prerequisites:
A role is chosen
Map layer is active

Stories:
1. A scale info bar is visible at the down-left corner of the screen
2. When the user opens the Tools menu by clicking on it, a menu opens. After this, the user clicks on 'Measure distance'.

The user can now click on the map. For each new point clicked the distance to the previous point is added to the total distance between all clicked points and is displayed in big letters.
When a closed loop (polygon) is created with the last point, the area spanned by all points is highlighted and can be classified as for example a no-go area.

Moving to location:

Prerequisites:
A role is chosen
Story:
User clicks on a 'Locations' button: A list of saved locations opens. Static and dynamic locations are stored in the high level database. User selects location from list: The map moves to saved location. If the map layer was inactive, it's activated beforehand. The zoom level is not altered. If a dynamic location (e.g. a moving robot) is selected, the map layer keeps focus on the centered location.

Indicator of status of actors:

Actor layer inactive:
Prerequisites:
A role is chosen

Story:
When the user clicks on the 'Actors' button, a list of actors with their current status is displayed. Actors can be

humans or robots. Possible status are for example: On-line, Busy, Off-line, Injured...

Actor layer active:
Prerequisites:
A role is chosen
Actor layer active

Story:
A colored label is shown on the pin at the location of each actor shown his descriptive status.

Database information properties
Prerequisites:
User/Actor is connected to the network

Story:
User inserts information which needs to be saved in the database (text, drawing, object position, image upload...).
TDS adds data to the database and adds a timestamp and the name of the actor.
After this all TDS instances involved in the TRADR system are updated with the new information (on map layer).

If the user has no connection he gets a warning message.
Nevertheless, the information on the TDS should remain visible even without a connection.
Also the edit dialog keeps the edited information.
The user can choose if he wants to discard the dialog or if he wants to wait until the connection comes back. If the connection is back, the information can be stored into the database.

Icon features:
Prerequisites:
A role is chosen
Map layer active
Icon is visible on the map layer

Informative:
Icon shows a thumbnail of it's icon image or some abstract graphics incl. status label for actors and other information.

Natural Scaling:
An icon should keep it's absolute size:
User zooms in on the map layer: Icons shrinks it's relative size
User zooms out on the map layer: Icons expand it's relative size

Icon locking:
User clicks on the icon: View expands and shows more information.
User clicks on the 'Lock' button: Icon toggles between locked in place and unlocked (default depends on icon type).

Field of view (FoV):
Icon shows orientation and field of view for robots and pictures if available.

Info rights management:
Prerequisites:
A role is chosen

Story:
A user only sees information the role is allowed to see.
Changed configuration of the appearence of the TDS will be stored into a local config file according to his role.
The next time a user chooses the same role on startup, settings from the previous local role session should be reestablished (e.g. opened layers, allocation and size of widget windows).

Layered structure:
Prerequisites:
A role is chosen

Story:
User opens layer overview: Shows a list of all layers with labels showing their status (enabled, disabled, unavailable/restricted...).
User clicks on a disabled layer in the overview: If user is allowed to open layer, add the layer to the interface and show it's content.
User clicks on a enabled layer in the overview: Information on that layer disappears.
User clicks on unavailable/restricted layer: System shows dialog, showing the user an error.

Faulty Connectivity Notifications:
No network connection
Prerequisites:
A role is chosen
Connection lost or database down

Story:
When the connection is lost, an according notification appears informing the user until the connection is available again.
No edits are possible at this moment.
All data of the layers stay visible.
When the connection is reestablished, the notification disappears and the user can enter data again and store it into the database making it visible for other interested parties.

Handheld device support:
Prerequisites:
A role is chosen
The application is installed on a handheld device(Tablet, phone) with a touch screen.

Stories:
Touching: The user can open menus/buttons by touching it, just like clicking on a computer.
Onscreen keyboard: When input from the keyboard is needed, an on-screen keyboard appears and can be used.
Multi-touch is not implemented at this point.

Multiple instances:
Prerequisites:
Multiple users are active

Story:
A user wants to join the current TRADR scenario.
The user can open the program, choose a role and see the same (role dependent) information as other users on other TDS instances having the same role.

Some information can be seen by different roles, some are only visible for some roles.

Display of Photos:
Show image and meta-info
Prerequisites:
Images available
A role is chosen

Story:
User clicks on a picture icon on the map layer: System shows image in image viewer. Next to image, available meta-information is displayed.
User clicks on the 'Exit image viewer' button: System returns to previous view.

BIBLIOGRAPHY

[1] *Tradr mission page*. June 23, 2015. URL: http://www.tradr-project.eu/?page_id=73.

[2] *Nifti Mission*. June 18, 2015. URL: http://www.nifti.eu/mission.

[3] G.J.M. Kruijff et al. "Designing, developing, and deploying systems to support human–robot teams in disaster response". In: *Advanced Robotics* 28.23 (2014), pp. 1547–1570. DOI: 10.1080/01691864.2014.985335. URL: http://dx.doi.org/10.1080/01691864.2014.985335.

[4] *Nifti Homepage*. June 23, 2015. URL: http://www.nifti.eu.

[5] Lionel C Briand and Jürgen Wüst. "Empirical studies of quality models in object-oriented systems". In: *Advances in computers* 56 (2002), pp. 97–166.

[6] *Style Guide for Python Code*. June 23, 2015. URL: https://www.python.org/dev/peps/pep-0008/.

[7] Beverly L. Harrison et al. "Transparent Layered User Interfaces: An Evaluation of a Display Design to Enhance Focused and Divided Attention". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '95. Denver, Colorado, USA: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 317–324. ISBN: 0-201-84705-1. DOI: 10.1145/223904.223945. URL: http://dx.doi.org/10.1145/223904.223945.

[8] *Marbe Homepage*. June 23, 2015. URL: https://marble.kde.org/.

[9] *ROS Distributions*. June 19, 2015. URL: http://wiki.ros.org/Distributions.

[10] *ROS-RQT Wiki*. June 18, 2015. URL: http://wiki.ros.org/rqt.

[11] *Python Wiki: Python2 or 3*. June 18, 2015. URL: https://wiki.python.org/moin/Python2orPython3.

[12] *Python Documentation: 28.11.future*. June 19, 2015. URL: https://docs.python.org/2/library/__future__.html.

GLOSSARY

binding A feature of Python is the ability to take existing libraries, written in C or C++, and make them available as Python extension modules. Such extension modules are often called bindings for the library. 8, 9

doctest The doctest module searches for pieces of text that look like interactive Python sessions, and then executes those sessions to verify that they work exactly as shown. 10

GUI Graphical User Interface. 1, 10

MapView A plugin to be created during this project as part of the TRADR Display System. It will display a map of the area which will be enhanced with points of interest. It will need to run alongside the OCU. 1, 6

MoSCoW The MoSCoW method is a method of ranking requirements on importance, so it is easy to understand. The consonants in the word stand for;
M: Must have,
S: Should have,
C: Could have,
W: Won't have
3

nose test A unit test making use of the Nose framework. 10

OCU GUI plugin build in RQT. It is used to control the robots used in the TRADR project and show the sensor data to the person controlling a robot. The OCU is part of the TRADR Display System and will run alongside the MapView. 1, 6, 8

RDF A family of World Wide Web Consortium (W3C) specifications originally designed as a metadata data model. 7

RedMine A project management web application including a wiki. 11

ROS Robot Operating System. 8, 11

rosdoc A tool that runs an external documentation tool (like Sphinx) on ROS packages. 11

RQT A ROS plugin, allowing users to create user interfaces on top of the ROS system. 8

SIG Company which performs dynamic source code analysis to ensure code quality. 6

Sphinx A tool allowing marked-up, automated documentation for Python and C/C++. 11

TDS Layered, modular display System for the TRADR system. 1, 3, 7, 11

TRADR EU-funded project to develop technology for human-robot teams to assist in disaster response efforts. 1

unit test A testing method used to test if a small unit of code (like a method) has the expected behavior in a single case. 10

Bibliography

[1] *Tradr mission page.* June 23, 2015. URL: http://www.tradr-project.eu/?page_id=73.

[2] *Agile Manifesto.* Aug. 15, 2015. URL: http://www.agilealliance.org/the-alliance/the-agile-manifesto/the-twelve-principles-of-agile-software/.

[3] *ROS Distributions.* June 19, 2015. URL: http://wiki.ros.org/Distributions.

[4] *RQT wiki page - section 5.2.* Aug. 26, 2015. URL: http://wiki.ros.org/rqt/Tutorials/Create%20your%20new%20rqt%20plugin.

[5] *Qt Homepage.* Aug. 27, 2015. URL: http://www.qt.io/.

[6] *Marble Homepage.* June 23, 2015. URL: https://marble.kde.org/.

[7] *RQT Wiki tutorial.* Aug. 26, 2015. URL: http://wiki.ros.org/rqt/Tutorials/Create%20your%20new%20rqt%20plugin.

[8] *Python Wiki: Python2 or 3.* June 18, 2015. URL: https://wiki.python.org/moin/Python2orPython3.

[9] *PyCharm.* Aug. 26, 2015. URL: https://www.jetbrains.com/pycharm/.

[10] *QT Creator.* Aug. 26, 2015. URL: https://wiki.qt.io/Category:Tools::QtCreator.

[11] *Sourcetree.* Aug. 26, 2015. URL: https://www.sourcetreeapp.com/.

[12] *Trello.* Aug. 26, 2015. URL: https://trello.com/.

[13] *Redmine.* Aug. 26, 2015. URL: http://www.redmine.org/.

[14] *Gitlab.* Aug. 26, 2015. URL: https://about.gitlab.com/.

[15] *overleaf.* Aug. 26, 2015. URL: https://www.overleaf.com/.

[16] *PyUnit.* Aug. 26, 2015. URL: https://wiki.python.org/moin/PyUnit.

[17] *Sphinx.* Aug. 26, 2015. URL: http://sphinx-doc.org/.

[18] *ROS-RQT Wiki.* June 18, 2015. URL: http://wiki.ros.org/rqt.

[19] *PyQT4.* Aug. 15, 2015. URL: https://www.riverbankcomputing.com/software/pyqt/intro.

[20] *Untangle.* Aug. 26, 2015. URL: https://github.com/stchris/untangle.

[21] *Sphinx autodoc usage.* Aug. 27, 2015. URL: http://sphinx-doc.org/ext/autodoc.html.

[22] *Nifti Mission.* June 18, 2015. URL: http://www.nifti.eu/mission.

[23] *MongoDB introduction.* Aug. 15, 2015. URL: http://docs.mongodb.org/manual/core/introduction/.

[24] *(Private) TDS TJEx-2015 prototype.* Aug. 27, 2015. URL: https://redmine.ciirc.cvut.cz/projects/tradr/wiki/TDS_-_TJEx-2015_prototype.

[25] Beverly L. Harrison et al. “Transparent Layered User Interfaces: An Evaluation of a Display Design to Enhance Focused and Divided Attention”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* CHI ’95. Denver, Colorado, USA: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 317–324. ISBN: 0-201-84705-1. DOI: 10.1145/223904.223945. URL: http://dx.doi.org/10.1145/223904.223945.

[26] Robert C Martin. “The open-closed principle”. In: *More C++ gems* (1996), pp. 97–112.

[27] *Stardog homepage.* Aug. 26, 2015. URL: http://stardog.com/.

[28] Lionel C Briand and Jürgen Wüst. “Empirical studies of quality models in object-oriented systems”. In: *Advances in computers* 56 (2002), pp. 97–166.

GLOSSARY

actor A person or robot with the task of working in the field. In the TDS the actors can be any member of the search & rescue mission and the robots (UAVs and UGVs). 2, 20

binding A feature of Python is the ability to take existing libraries, written in C or C++, and make them available as Python extension modules. Such extension modules are often called bindings for the library. 6, 7

dostring A string literal in the source code to create documentation or comments. This literal will not be stripped during compilation, therefore they are accessible during runtime for referencing or meta-data.. 7, 8

GUI Graphical User Interface. 3, 10, 13, 21

HTML A markup language which allows the usage of hypertext/links to other documents. It is mainly used for webpages. 8

MapView A plugin created during this project as part of the TRADR Display System. It displays a map of the area which is enhanced with points of interest. It runs alongside the OCU. 3

MarbleWidget A widget in the Marble module, showing the map aspect of the Marble user interface in a compact and adjustable widget. 6, 23

MoSCoW The MoSCoW method is a method of ranking requirements on importance, so it is easy to understand. The consonants in the word stand for;
M: Must have,
S: Should have,
C: Could have,
W: Won't have
11

OCU GUI plugin build in RQT. It is used to control the robots used in the TRADR project and show the sensor data to the person controlling a robot. The OCU is part of the TRADR Display System and runs alongside the MapView. 3, 7, 10

POI Point Of Interest. 1, 23

polling Checking the database at a defined rate to check for new or changed information.. 23

QWidget A normal widget provided by QT with specific functions. See also Widget. 21

ROS Robot Operating System. 6, 7

RQT A ROS plugin, allowing users to create user interfaces on top of the ROS system. 1, 6, 7, 15

S&R search and rescue. 2

SIG Company which performs dynamic source code analysis to ensure code quality. 1, 26

signal A message raised as response to a (user) action. A signal can be used to call slots after for example a button is clicked in the user interface or if an value changes.. 16, 21

slot A function that can be called by an Qt signal.. 21

TDS Layered, modular display System for the TRADR system. 1, 3, 6, 13, 15

TJEx A testing moment of all the systems created by the TRADR consortium done by simulating an urban search and rescue mission.. 10, 11

TRADR EU-funded project to develop technology for human-robot teams to assist in disaster response efforts. 1, 2, 9, 15

UAV Unmanned aerial vehicle. 2

UGV Unmanned ground vehicle. 2

USAR Search and Rescue in urban or industrial environments, mainly aimed at confined spaces such as collapsed buildings. 2

widget (Small) Element of a user interface. Can often be seen as a small program within another program.. 21

XML A markup language to create structured human- and machine readable documents. 24