Viola project

Morten Silberbauer Sabinsky

Kongens Lyngby 2006

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

This thesis describes how a "State of the art" framework for embedded systems can be designed. The target for this framework is a transcutane monitor also known as a TCM device. A TCM is a device capable of measuring blood gasses, pulse rate and oxigen saturation through the skin of a patient, without piercing the skin. The goal of the thesis is to create a framework consisting of a number of self contained software modules each capable of carrying out one specific task. Emphasis have been put on the usage of design patterns and usage of the most current theories regarding designing embedded frameworks. This means theories of software evolution, composite design patterns and slippage problems has been used in the framework design. The thesis explores how some of these theories can be used to create new design patterns and use excisting patterns to combining several small patterns into a new larger pattern.

The software modules designed in this framework cannot communicate directly with each other. For this reason a software module have been created to handle communication between software modules. Also modules controlling how objects are shared, how user the user interfaces are controlled and how integrity is kept in XML based configuration files.

The thesis is separated into 4 main chapters.

Chapter 1 is designed to give the reader a description of the target TCM system, the overall goals for the thesis and a preliminary analysis of the requirements and concepts to be used. Lastly there is a small description of the challenges solved before the design was created.

Chapter 2 describes each module designed during the thesis. The modules

describes methods for sharing objects between the modules using a factory sphere, methods for using a unified way of communication using a communication manager, method for controlling user interfaces using an input controller and configuration files using a configuration manager.

Chapter 3 shows how the designed modules can be used together to create a TCM device and how to solve some of the problems the given design gives.

Chapter 4 contains the conclusion of the thesis and a list of items where improvement could be done and areas that should be included if further development should be done.

Resumé

Dette speciale beskriver hvordan et "State of the art" framework til et embedded system kan designes. Det beskrevne framework er designet til køre på en transcutan monitor, i dette tilfælde et TCM apparat. Et TCM apparat bruges til at måle blodgasser, puls og iltmætning igennem en patients hud, uden at prikke hul i huden. Målet for dette speciale, er at lave et framework, som består af små selvstændige software moduler, som kan løse en specifik opgave. Vægten i dette speciale er lagt i brugen af design patterns og det nyeste teorier for design af embeddede frameworks. Det betyder at teorier om software evolution, sammensatte design patterns og udvidelsesproblemer er blevet brugt i designet. I specialet vil der også blive set på, hvordan nye typer design patterns kan laves og hvordan små design patterns kan kombineres til større og mere komplekse design patterns.

De software moduler der er designet i dette framework, kan ikke kommunikere direkte med hinanden. Det betyder at de kræver hjælp fra andre moduler og derfor er der lavet software moduler til at styre kommunikation, objektdeling, brugerinterface og integritet af XML baseret konfigurationsfiler.

Specialet er delt op i 4 hovedafsnit

Kapitel 1 giver læseren et overblik over TCM systemet, de overordnet krav til og begrænsninger for specialet. En tidlig analyse af specialet er lavet for at få et overblik over koncepter og krav. Sidst er der en kort beskrivelse af en række udfordringer som er blevet løst for at give et bedre grundlag for designet.

Kapitel 2 beskriver de software moduler som er blevet designet. Software mod-

ulerne beskriver metoder for deling fabriksobjekter mellem software moduler, metode til kommunikation mellem software moduler ved brug af en kommunikations manager. Dertil metode til at styre views ved brug af en input controller og metode til at styre konfigurationsfiler.

Kapitel 3 bruges til at vise en række eksempler på TCM relateret brug af det implementerede framework, samt hvordan nogle af modulerne i samspil kan løse nogle af de beskrevne problemer.

Kapitel 4 indeholder specialets konklusion og lister en række punkter, hvor forbedringer er nødvendige hvis frameworket skal bruges.

Preface

This thesis is a result of the Master of Science project done in collaboration between IMM/DTU[1] and Radiometer Medical ApS[2]. The thesis has been made by Morten S. Sabinsky student no. S973936 and have been supervised by Bjarne Poulsen at IMM/DTU and Jørgen Belfalas at Radiometer Medical ApS, TCM development.

The purpose of this cooperation is to bring new knowledge and demonstrate new technology to Radiometer that could be used for the next generation of TCM devices. None of the methods and theories in this thesis has been used for TCM devices before, making this a new area of software design and development for Radiometer.

The thesis was original called "Modular TCM Application" it was however in the beginning of the thesis given the codename "Viola" by Radiometer. This name has been used since then.

It is recommended the reader of this thesis have some experience with design pattern, understands UML diagrams and some basic knowledge regarding C# and the .NET 2.0 framework.

[1] Richard Petersens Plads. DTU - Bygning 321. 2800 Lyngby. reception@imm.dtu.dk. Tlf. 4525 3351

[2] Radiometer Danmark. Åkandevej 21. 2700 Brønshøj. Tlf: 3827 2829. Fax: 3827 2712.

Acknowledgements

First I want to thank Radiometer for allowing me to make this exciting thesis. I thank all who have supported me during the period of this thesis and to all of you who have been reviewing the thesis with me.

My thanks go to
Counseling
Bjarne Poulsen IMM/DTU
Jørgen Belfalas RMED TCM development

Proofreading
Michael Ditzel RMED TCM Development
Ole Hansen REMD TCM Development
Oluf Dannevang RMED TCM Development
Thomas Gerken

Contents

Chapter 1

Introduction

Contents

This chapter is used to give some background information's about the TCM devices, method and tools used. Books, papers and links are shown in the bibliography and a small description of each is given this chapter. It also contains the preliminary analysis done before the design. This analysis was done to plot areas that needed to be probed before the design could begin. This analysis was also an important part in establishing a time schedule for this thesis.

1.1 Thesis vision

The purpose of this thesis is to explore a modular software design where 3 TCM variants can be combined into one application. The design should be easy to evolve, maintain and allow reuse of components. In this thesis the main focus will be the design of a framework to support a modular software system. The framework should be as general as possible and to show how it can be used a number of cases will be described.

1.2 About Radiometer

Radiometer Medical is a company that is specialized in devices that can measure different parameters in blood. Currently they are one of world biggest companies in this area with a global marked share around 50%. It was founded in 1935 where they developed measuring devices for the radio industry. They started producing medical devices in 1954.

There are 2 main types of blood measuring devices developed by Radiometer. The main product type if called ABL. This is a device that measures directly on blood samples and is capable of measuring a large number of parameter. Some of these are pO_2, pCO_2, cK^+, cNa^+, $cGlu$, $cLac$. More can be seen at http://www.radiometer.com/

The second type of device is called TCM and these devices are the once this thesis will concentrate on.

1.3 TCM devices

TCM stands for Transcutane Monitor. This type of device is used to measure blood gasses through a patient's skin. This is a noninvasive way of measuring patient vitals and is therefore used for long term measuring to determine trends.

The current TC-Monitor version is 4 and it comes in 3 variants. These are TCM4, TCM40 and TCM400.

The hardware platform for these variants is mostly the same. The Hardware platform consists of a base unit where up to 6 hardware-measuring modules

can be attached. Depending on the hardware configuration there are 2 different software packages. These are a TCM4/40 package and a TCM400 package.

The platform currently uses Windows CE 4.2 as a operating system.

1.3.1 Device overview

TCM4

This device is capable of measuring O_2 and CO_2 using a single sensor that combines the parameters.
This device is mainly used in neonatal care.
This device consists of a TCM4/40 software package and a hardware module capable of measuring O_2 and CO_2.

TCM40

This device is capable of measuring O_2 and CO_2 using a single sensor that combines the parameters. It can also measure pulse in beats/minutes and SpO_2 using a second sensor.
This device is mainly used in sleep labs when treating sleep disorders.
This device consists of a TCM4/40 software package and 2 hardware modules. The first module can measure O_2 and CO_2 and the second can measure SpO_2 and Pulse.

TCM400

This device is capable of measuring O_2 using up to 6 independent sensors.
This device is mainly used in wound treatment.
This device consists of a TCM400 software package and between 1 and 6 hardware modules capable of measuring O_2.
This allows for measuring O_2 in up to 6 different places on the body.

Common for all of the devices is that measurements for O_2 and CO_2 can be displayed using mmHg or kPa.

1.3.2 TCM4 History

The first TCM4 variant was the TCM400 and it was released around the year 2000. Then came the TCM4 variant and a later extension allowed for a TCM40

variant. The software for these variant has been continuously updated the last 6 years to keep up the new demands for functionality.

This has evolved the application far beyond what the original was designed for and resulted in an application that is complex and difficult to maintain. It has also been through several hardware versions and Windows CE versions requiring small modifications to the application.

1.3.3 Problems with the currents design

The current applications are not only difficult to maintain but they are also difficult to extend. Extending the applications to support new parameters or sensors requires changes to large portions of the application. This requires a lot of retesting before the applications can be released.

The 2 software packages also come in up to 12 different versions depending on the language selected.

1.3.4 Future TCM

How a future TCM device should be is currently not known. This makes it challenging to design a framework that could support it. However it is know a future system would support the current parameters and probably more. To start with the thesis should take the current technologies and use these as a foundation but also keeping in mind that this could change. Every thing discussed in this thesis has never been done before for a TCM device and should be considered as an evaluation model.

1.4 Preliminary analysis

To get at overview of the Thesis a small analysis is required. This analysis is used to get an overview of the technology required, the methods and tools to use. It should also be used to generate some concept ideas to be used later in the design process.

The section contains:

- Description of the technology used.
- Description of the tools used.
- Description of methods used.
- Description of some of the most current theories used.
- Description of some concepts to be used when designing a software module.

After reading this section an overview of the methods and theories used, should have been gained. Also a preliminary system has been given and will be used as a guideline for the final system.

1.4.1 Technology

The software development will use compact framework 2.0 and windows CE 5.0. The reason for selecting compact framework 2.0 is its support for communication with serial ports. This new addition to the framework makes it attractive for Radiometer to investigate possible usages of .NET. The size of the framework is also a factor since the software has to run on a small-embedded platform with limited space for storing applications and data.

An alternative to compact framework 2.0 is OpenNETCF[1]. This is an open source version of the compact framework. The reason this framework has not been taken into consideration is concerns regarding license requirements. If this framework is to be used then the legal department has to explore the license requirements first to give an overview of any license problems.

The reason for selecting Windows CE 5.0 is that it is the only platform currently supporting compact framework 2.0, for customized Windows CE images. If this

[1]See more at http://www.opennetcf.org

project has to be used commercially later, the price per license is also an issue. Using Windows CE the price for a license is about $4 to $20 per device, were an alternative OS like Embedded XP is about $70 to $80 per device.

Embedded XP have been disused as an alternative usage to Windows CE the reason for this is it can support the full .NET 2.0 framework. The reason for not selecting this product is primarily its size and resource usage. Windows CE have an image footprint of about 16Mb where Embedded XP is about 300Mb.

Open source operating systems like BSD and Linux has not been considered as a viable solution. This is primarily because license concerns regarding source code, application and time consumption trying to implement a system like this. Although this could be an interesting project later on since most can be freely distributed saving the license cost to Microsoft.

1.4.2 Tools

Visual Studio
Visual Studio 2005 can be used for developing the software. It can also be used to generate some of the simple class diagrams needed.

Platform builder
Platform builder 5.0 is used to create Windows CE images to test the application on.

CE Emulator
CE 5.0 Emulator is used to launch generated Windows CE images.

Visio
Visio 2003 will be used to create drawing to support the documentation. These are sequence diagrams, class diagrams and all other drawings. The reason for using Visio for class diagrams is because the class diagram system in Visual Studio cannot show all the necessary information's to explain relations.

LaTeX
LaTeX will be used to generate the thesis documentation. This is a requirement from DTU. Delivery information's can be found at:
http://www2.imm.dtu.dk/teaching/thesis/

TeXnicCenter
TeXnicCenter is an IDE for LaTeX. This will be used for writing the thesis in LaTeX. Information about TeXnicCenter can be found at:
http://sourceforge.net/projects/texniccenter/

1.4.3 Methods

Iteration

When developing the software an iterative method will be used. The method will follow the procedure

1. Analysis and requirement specification
2. Design
3. Implementation
4. Test
5. Summery

If the summery says the result is not satisfactory or some of the requirements are not met, then the procedure starts again using the current result as a basis. Iterations can skip 4 and 5 if a design flaw is discovered during the implementation. This should save time in the overall development process. When skipping steps is should be considered if the summery could have any value. Describing the design flaw might help avoiding the same pitfall at a later iteration.

To limit the time spent on a specific part of the application there should not be performed more than 4 iterations in each part.

Design pattern

Design patterns are predefined software structures describing common functionally used in applications. This could be functionality describing object creation and how object interacts. They should be used when possible. Not meaning to force the used of design patterns but using the where it makes sense.

UML

To document the application UML diagrams will be used, to support the descriptions. Class diagram and sequence diagram will mainly be used. Class diagrams will be used to give an overview of interface and class relations. Sequence diagrams will be used to give an overview of complex functionality.

The reason for using UML to support the documentation is because Radiometer Medical uses UML in their development process.

1.4.3.1 Summery

Chapter 2 contains the design of the system. Each main section describes a specific part if the system using the iterations described. This means each section self contained from the rest of the design.

The methods and tools described are only a part of what is used. The next section describes the theories used.

1.4.4 Analysis of current theories

This is a part of the method analysis. The reason for this is because most of what is found in the books and papers will be the basis for the method used in this thesis. The purpose of this section is to shortly describe the papers used containing some of the newest knowledge in the areas of interest.

1.4.4.1 Books

Design patterns [1]
This book describes commonly used design patterns. I have chosen this book because the description of the patterns are good and it also describes when it is most beneficial to use the patterns. The patterns described in this book are most commonly used to implement specific functionality. I have mostly used this book as a library to lookup descriptions of patterns I needed.

Patterns of Enterprise Application Architecture [2]
This book is used to describe how to implement enterprise systems and design patterns used in enterprise applications. Although most of the designs are for large system and not embedded system, there are some patterns that are interesting for this thesis. These are patterns describing data storage, plug-in systems and user interface control.

1.4.4.2 Papers

How to tackle the Slippage Problem in object systems [3]
This paper describes methods for handling slippage problems and methods for avoiding slippage problems. A slippage problem is a situation where a piece of software has to be extended beyond what the design is capable of. As an example this could be an added parameter to an interface function, this often requires changes to large part of the software. What will be used from this paper are the parts that describe the methods for avoiding slippage problems.

Adaptive Plug-and-Play Components for Evolutionary SWD [4] [2]

[2] SWD : Software Development

This paper discusses methods and techniques for creating modular software solutions and methods for evolving these. Mostly the techniques for modular systems will be used from this paper. However the description of how to evolve the system could be interesting to look at, at a later date. Especially if evolution process could be combined with the slippage problems.

Composite Design Patterns [5]
This paper discusses how to combine design patters to create new larger design patterns. This is interesting since combining design patterns can solve complex problems. The guideline described should to help avoid the pitfalls described when designing the software.

A Layered Architecture for Uniform Version Management [6]
This paper discusses administrative methods for controlling software versions for both small applications and large component based system. It also gives some insight into the tools that could be used to keep track of versions. What are interesting in this paper are the methods used. Some of these could be implemented into the software itself.

A Simple and Practical Approach to Unit Testing [7]
This paper discusses methods for unit testing software. It is especially design for Java and it also discusses some of the tools available. The methods described can however be transferred to any system.

Patterns as Signs [8]
This paper discusses methods for identifying patterns in software and tries to give a definition of what a design pattern is. This is interesting since it in combination with Composite Design Patterns paper can be used to verify the end result.

1.4.4.3 Links

Wiki [9]
This site contains links to different sites containg informations about design patterns of different types and methods to implement these.

1.4.4.4 Summery

There is not much documentation describing framework design for embedded systems. Most of what can be found is for large enterprise systems. This means that most of the methods described in the books and papers cannot be directly

used in the form they have. However they can be used for inspiration, to make lighter version of the methods and patterns, and to help avoid some of the pitfalls described in the papers.

1.4.5 Concept ideas

These ideas came up during a brainstorm and sorted for usability. This means these concepts are the ones that were considered to give the best preview for the components in the desired system.

1.4.5.1 Concept 1 HAL

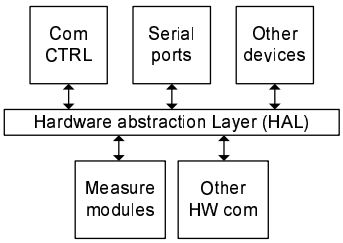

Figure 1.1: Hardware abstraction layer

A hardware abstraction layer (Will be referenced as HAL) is needed to keep weak bindings between the software and hardware. This layer hides the hardware specific information from the software above. This ensures the software functionality if the under laying hardware is changed. The concept idea is shown on figure 1.1

1.4.5.2 Concept 2 Core components

A modular software design needs a core system to support the modules. This core system or framework consists of several sub components, each mandatory for the application functionality. The concept idea is shown on figure 1.2

HW discovery
: This component is used to identify connected hardware modules. Once all hardware modules have been identified, it selects a configuration file describing the modules that is needed.

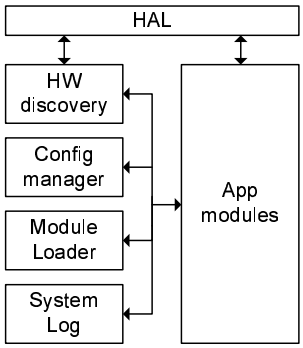

Figure 1.2: Core components

Config manager
This component is to handle global application configuration and local configuration for each module. Global configurations could be license restriction and default data. Local configuration could be alarm limits or ranges selected for displayed curves.

Module loader
This component is responsible for loading application/software modules, verify integrity and initialize them.

System log
This component is responsible for storing log information for the entire system in a common format. Log information's could contain information's about errors or other event important for the overall system operation.

App modules
These components are software modules loaded to make up the functional part of the application. These software modules could be modules used to measure O_2 or SpO_2 values and modules to display O_2 or SpO_2 values.

1.4.5.3 Concept 3 Module layering

Layering of the modules is needed to give a functional overview. Modules are to be separated into 3 layers seen on figure. 1.3

Layer 3
This layer is to contain user interface software modules.

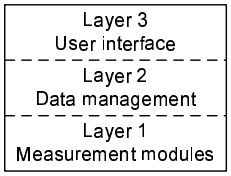

Figure 1.3: Module layers

Layer 2
This layer is to contain software modules responsible for data management. This is storing measurement, exporting/importing data.

Layer 1
This layer is to contain measuring software modules.

1.4.5.4 Concept 4 Resources

Software components often need software resources like text strings or images, especially if the system needs to display text in different languages. To simplify resource management only software modules in layer 3 may have language dependent resources. No software module should share a software resource.

1.4.5.5 Concept 5 Events

Since this application is build up of many small software modules it would be beneficial to have a system to handle events. This event system should be able to dynamically add and remove events. Software module requiring an event should be able to add a listener to an event.

1.4.5.6 Summery

The concepts given describe a modular system and this should be the goal of this thesis to create a system. The concept does not give solutions that can be used directly, but they give inspiration and a direction for the design process.

1.5 Challenges

Before design of the modules can begin a number of challenges have to be solved. These challenges are done to highlight and eliminate certain problem areas, which exist due to lack of knowledge.

The challenges that have been looked at in this thesis are:

1. Platform image.
2. Connectivity between Visual Studio and CE.
3. Serial connection.
4. Event handling.
5. Software module integrity.
6. Third party software modules.
7. Module hardware interface.
8. Loading modules Runtime.
9. String localization.
10. Modules version control.
11. Integrity of configuration files.
12. Serializing objects.
13. FDA and CE approval.

Appendix I gives a detailed description of each challenge and the results.

1.5.1 Summery

Most of the challenges solved were relative easy once the problem was specified. However there were some problems that took a lot longer to solve than expected.

Creating a custom CE device to the current hardware platform proved impossible with CE 5.0. The problems found might have been solved if more time had been available. A bug in the CE 5.0 was discovered when creating images to

at PC. COM1 was always locked to send out debug information's, even though the loader was told to shut down the debug system. A solution was found but not tested due to time restrictions. The image used, uses the CE emulator. The solution is acceptable but performance cannot be tested in this environment.

Creating a connection between a Windows CE and Visual Studio 2005 (VS) was a lot more difficult than expected. VS depend on the Active Sync application to handle the communication. This makes it very easy to develop applications to handheld devices. But VS has almost no support for developing applications to custom CE device. A guide in appendix B describes what have to be included in the custom CE device and all the manual steps that have to be carried out to establish a connection.

1.6 Test strategy

A test strategy is necessary to insure the quality of the application. This has to be sketched out before the application design starts, in order to incorporate testable requirements. Inspiration for the test strategy have been found using ref. [7].

Unit testing ensures a uniform test of objects but can also be quite difficult, especially if there are many similar objects in the system. This is because normally there is one unit test for each object. This causes repetition of test steps. To reduce the complexity to the unit test, it will be split into 2 levels for each object.

Level 1 test

The purpose of this test is to verify implemented interfaces. These tests are common for all objects using a specific interface.

Creating a standardized interface test reduces overall complexity for the unit testing system. Instead of repeating steps in several unit tests, it can now reuse the same test for several objects. Creating a standardized test however does have some problems. Firstly it cannot test all functionality in an object, secondly the interface have to have a strong specification, describing all input and output, and reactions to failure.

This type of test can be implemented into the application itself. This could help development of object by letting the application test the object before it is initialized the first time. Also it can help third party developers because they do not need to have a test environment for the objects.

Level 2 test

The purpose of this test is to test functionality not covered by level 1 test. This type of unit test uses an external test environment designed for the specific object.

Non-unit test

Not all functionality can be tested by automatic unit tests. Most of these tests are tests requiring user input or visual inspections.

These are often objects used to display user interfaces. Testing these objects can be done, by creating test stubs simulating the underlying system. Simulation is preferred since this can control all underlying functionality and thereby allowing the test to get into all states of the user interface.

Integrations test

Integrations tests are used to verify the interaction between several objects/software modules. This can be the final application or parts of it.

These tests should be relative simple tests since the interfaces used for interaction have been tested.

1.6.1 Test documentation

To document how a test is performed a step-by-step list has to be made. This list should describe requirement/functionality tested and expected result. The test description must include the test type, the interface or object tested and test results for each step. A template for a test protocol can be found in appendix G.1. Once a test protocol have described for an object, a review of the protocol must be performed. The purpose of this review is to validate the protocol against the objects requirements and design. This should revile any flaws in the test protocol.

If an error is found in a objects that have passed its test protocol, a new review should be made of the requirements, design and test protocol. This is to revile where the errors come from and where changes have to be made, so this error can be caught if it should occur again.

1.7 Overall requirements and limitations

The overall requirements have been created from the information given in the thesis description in appendix D and the information gained during the challenge analysis. The overall requirements and limitations are used to create a preliminary time schedule.

Specific requirements for each module are identified in the design for each module.

1.7.1 Overall requirements

OR1	The framework must be module based.
OR2	The modules must be signed.
OR3	The modules must be identifiable with a GUID value.
OR4	Events handling must be done using delegates.
OR5	Module must be loaded using reflections.
OR6	Traceability must be implemented into the thesis documentation.
OR7	The framework must be developed in Visual Studio 2005.
OR8	The framework must be implements using Compact framework 2.0.
OR9	The framework must be executable on a Windows CE 5.0 emulator target.
OR10	The framework must be implemented using the language C#.
OR11	Test strategy must be described.
OR12	Software modules must use a serial port for communication with hardware modules.

OR4 events using delegates

Challenge 4 was used to test different ways to perform event handling. Primary focus should be put on using delegates to perform the event handling. The reason for this is delegates seems to offer the most flexible functionality.

OR6 Traceability

This is a requirement given by FDA. Traceability is to give the reader a chance to see where requirements are described, implemented and tested in the documentation done for a project. In this thesis all requirements have been given an Id. Sections that describes requirements will have a reference to the requirements.

1.7.2 Limitation

The main focus of this thesis should be in the design of the framework software modules. Software modules for user interfaces, data storage and measurement should be secondary. They should only be implemented in a limited version to evaluate the framework functionality.

The main focus on the framework should be to develop the methods in the following order:

1. Method for object sharing between all the modules.
2. Method for event handling between the modules.
3. Method for data transfer between modules.
4. Method for controlling user interfaces.
5. Method for controlling framework configuration.
6. Method for loading modules runtime.
7. Method for separating modules from communicating directly with hardware.
8. Method for discovery of hardware modules.
9. Method for controlling languages in the user interface.

All methods from 5 to 9 are only to be made once there is a satisfactory result of the previous items is established. The items should therefore not be outlined in the preliminary time schedule. If these items make it into the thesis they should be shown in the final time schedule in appendix H.2.

A test strategy must be described however there should not be implemented a test for every module created. Test should only be implemented to demonstrate and verify some of the main modules functionality.

1.8 Introduction summery

This chapter have been used to give some background information about TCM devices and their usage. Information's about tools and methods have been given.

A number papers and books have been shortly described. These describe some of the most current theories available. Theories for making modular systems for embedded devices are almost non-existing. Most of the theories found have their main focus in large applications and enterprise system. This means most of the theories cannot be used directly in this thesis. They can however be used for inspiration to make new theories and designs. The chapter contains descriptions of a number of challenges. These challenges have been identified and solved to help establish a foundation for the design of the framework.

The next chapter will concentrate on designing modules for the framework. The chapter will mainly focus on modules to the framework. The modules will be designed using the iterative method described. This means each section describing a module contains analysis, design and implementation.

Chapter 2

Analysis of core architecture for future TCM device

Contents

This chapter contains several sections each describing a software module in the framework architecture. The chapter has the following structure.

1. A short overview is given, to show all the components in the framework and a small description of where focus have been put in the framework
2. A section describing the basic architecture of the software modules used in the framework.
3. A section describing a method for sharing factory objects between software modules without the software modules know each other.

4. A section describing a method for software modules to communicate with each other.
5. A section describing a method for controlling configuration files and ensuring the integrity of these
6. A section describing a method for controlling views in a user interface.
7. Lastly a small summery of what have been archived and a discussion of whether some of the solutions could be seen as a new type of design patterns.

2.1 Preliminary architecture overview

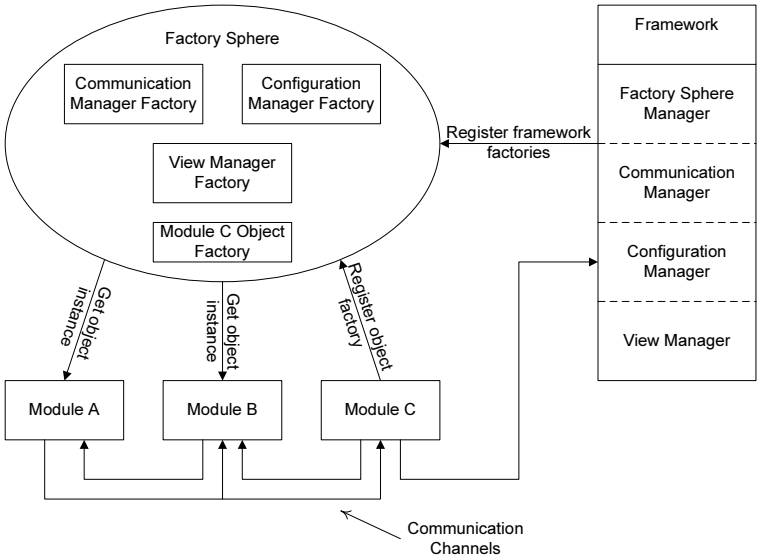

Figure 2.1: Preliminary architecture of framework

Figure 2.1 shows the architecture for the system to be designed.

The system has 3 main components. To the right is the framework block. This is the block where the main focus will be put in the design. The purpose of the framework block is to contain a number of modules capable of controlling and support the external software modules. The software modules shown in the right block will be described in the coming sections.

The middle contains a sphere. The main purpose of this sphere is to contain factory objects capable of creating objects used and shared by the framework and external modules. The factory sphere is a virtual container used by the framework.

The bottom shows 3 external software modules. These modules could be measurements modules measuring O_2 and CO_2. They could also be module used to store data and user interface components. External modules will loosely be described but will not be designed.

2.2 Software modules

The purpose of this section is to describe an overall design for software modules in the framework architecture. This means the design should be used for both framework software modules and external software modules. The section contains.

1. An analysis describing the overall module design and desired structure.
2. List of requirements for a basic software module.
3. Information about how versions should be handled.
4. Information about how integrity of software modules should be tested.
5. An interface design for software modules. This interface should be the only part of the software modules, exposed of the outside.
6. Some guidelines for initialization and configuration of software modules. These are only overall guidelines. Specific information about configuration located in the design of each software module module.

2.2.1 Module analysis

Software modules are the basic components in the framework architecture. These components are used in the framework core and as plug in modules.

A software module can be seen as a UML package that contains on or more objects. The defined software module interface is the entry point into the package.

Figure 2.2 shows a concept idea on how software modules should be build. The software module interface should be as minimal as possible, removing all operation-oriented functionality from the interface. By operation oriented functionality it is ment all access to specific functionality in a module should not be allowed. Communication between the modules through the module interface should not be allowed. The reason for these restrictions is to minimize bindings between the modules. The module interface should however contain a function to set a reference to a global factory system, allowing object to be shared in a controlled manner between the modules. Also a function to initialize the module should be present. Inspiration for this design comes from ref. [3] and [4]

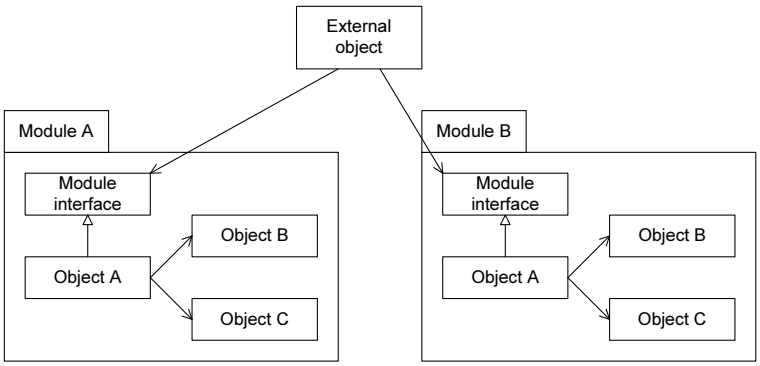

Figure 2.2: Module concept

2.2.1.1 Design qualities

- No binding between modules.

2.2.1.2 Design drawback

- No way to interact between modules.

2.2.1.3 Summery

This design will generate modules that can function by them self, with no software binding to other software modules. This type of software module can then be used as building blocks to make a larger system. The only function a module should have is an initialization function (MIR2 found in section 2.2.2).

2.2.2 Module interface Requirements

MIR1	Module must have a unique GUID value to identify the module.
MIR2	Module must have an initialization function.
MIR3	Module must have a reference to the factory sphere.
MIR4	Modules must use strong name signing.

Requirement MIR4 can only be fulfilled during compilation of the module. Hence this is something that has to be setup in the project itself.

2.2.3 Module identification

Software modules should be uniquely identifiable to ensure the software modules are of a valid type and not loaded more than once. Only allowing software modules to be loaded once limits the overhead in integrity testing modules. A GUID value has been selected to identify software modules. A GUID value is a 128bit value usually generated using a random generator. This gives with high probability a unique value.

Using these unique values to identify a software module can not only be used to identify software module type, but also a specific software module version. These values can be put into a database linking them to module type and version. This gives us the ability to control module types, versions and owners.

Instead of using random numbers in the GUID value a system could be used by splitting up the 128bit value into sections, type, version and owner. Splitting up the values like this does however create some limit on the amount of modules for the system. But it can also give a better overview of the modules and to a limit remove the need for an external database of the software modules.

2.2.3.1 Design qualities

- Gives a unique Id of module and version.
- Gives a way to administrate modules and versions.

2.2.3.2 Design drawbacks

- Requires administration of modules and versions.

2.2.3.3 Summery

The use of GUID values should be added as a requirement (MIR1). The administration of GUID values is seen as a drawback and a quality of the design. The reason for this is administration can be time consuming and cost money, but it gives a way to control the modules. GUID's must be implemented in this thesis, but methods for version control will not be discussed.

2.2.4 Module integrity

Compact Framework 2.0 supports strong named assemblies. Strong named assemblies are assemblies there have been signed with a private RSA (ref. [10]) key and with the public key embedded into the assembly. The RSA key pair can be supplied through a simple key pair file or through a Personal Information Exchange certificate (similar to X.509 certificate ref. [11]). If an assembly is signed then the compact framework will automatically verify the signature. If the verification fail then the loading of the module will be aborted. If software modules are static linked then all software modules have to be signed. Software modules however do not need to use the same key pair. This gives the possibility to have different key pairs to different module suppliers. By creating a list of valid third party public RSA keys and compare these keys to the module key, then it is possible to control access to supplier's modules. This will however require some form of updating of the key list. Another solution could be that all modules have to se signed using the same key. This will require some type validation procedure from Radiometer. One problem to consider when selecting method, is that the signing procedure for strong named assemblies, is done during build time.

2.2.4.1 Design qualities

- Gives a strong integrity verification.
- Gives the ability to verify module owner.
- Gives the ability to control allowed third party modules.

2.2.4.2 Design drawbacks

- Gives a longer load time for modules.

2.2.4.3 Summery

The only drawback when using strong name signing is the load time. This is increased since modules must be verified before they are executed. This however only affect the time it takes to start the application. Once the application is started it has no effect on performance.

Strong name signing must be added as a requirement (MIR4).

2.2.5 Module interface design

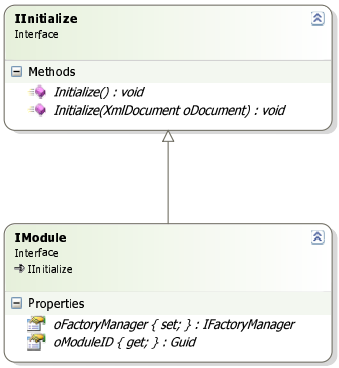

Figure 2.3: Module interface design.

Figure 2.3 shows how the software module interface is designed. The interface has 2 properties. The module interface does also have a small initialization interface. This interface is used to initialize modules and also certain objects.

oFactoryManager
: Property is used to set a share a global abstract factory system for object sharing. This is specified as a requirement (MIR3).

oModuleID
: Property used to get a GUID value used to identify a software module (MIR1).

Initialize
: Function used to initialize a software module (MIR2).

Figure 2.4 shows how a module is initialized.

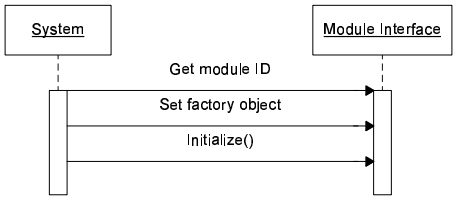

Figure 2.4: Sequence for module initialization.

2.2.6 Initialization and configuration guidelines

2.2.6.1 Initialization

This section describes some guidelines modules should follow when initializing software modules. The initialization of software modules should be done in 3 stages.

Stage 1
First a module should get the configuration it needs to set itself up. Once the configuration have been retrieved it should be checked for missing data and if data follow the setup requirements. As an example if a limit is set higher than allowed.

If the configuration cannot be verified as valid a default configuration should be used instead. A note in a log system should also be set, saying the retrieved configuration was not valid.

Stage 2
Required objects located in the factory sphere, should be retrieved and if necessary initialized.

Stage 3
Threads and listeners should be initialized.

2.2.6.2 Configuration

This section describes some guidelines modules should follow when it is configured. In section 2.4.2 a method for communication using channels between software modules is described. To make a software module as flexible as possible, the identification for the input and output channels should be described in a configuration file. This would allow for redirection of data without recompiling the software modules.

As an example a system can have two channel, channel A and channel B. Channel A show data on the screen and in this system is used for debugging. Channel B is used in the final system and stores data on a disk. When working on a module or interaction between modules the system can be configured to send data to channel A to see if data is correct. Once data is verified then the configuration can be changed to channel B storing data instead. If no software changes have been performed on the software module then test of the modules is not necessary. However if channels were hard coded into the modules. It

would then be necessary to test the module every time a change was done. The ability to redirect data in the configuration should also make it easier to reuse a module.

2.3 Factory sphere system

The purpose this section is to give the description on how factory objects can be shared between software modules in the framework and external software modules. The design given tries to make a new type of factory system, which is similar to an abstract factory.

1. The first thing given is a small description of where the factory system fits into the framework architecture.
2. The analysis starts with a small description of the abstract factory pattern and lists some for the qualities and drawbacks of this design.
3. A prototype design of the factory sphere is created to see if the qualities of the abstract factory can be maintained in a system that can be configured at runtime.
4. A front-end prototype for the factory sphere is created to se if this can solve some of the problems that were discovered when Implementing the prototype factory sphere.
5. After the analysis and prototypes a list of requirements are given.
6. A final design is described, this design contains information about interfaces used, how generic factory objects can be made, how objects created should be initialized and how creational patterns can be implemented as generic factory objects.
7. An overview of the objects and interfaces designed, and relations between these are given.
8. A summery for the factory sphere system.

2.3.1 Architecture overview for Factory Sphere

The highlighted components in figure 2.5, shows where the factory sphere manager fits into the framework architecture. The factory sphere manager module is a part of the framework and it is responsible for controlling how software modules can share access to factory objects.

As shown in the interface design for the module in section 2.2.5, each module contains a property to set a reference to the factory sphere. This means all

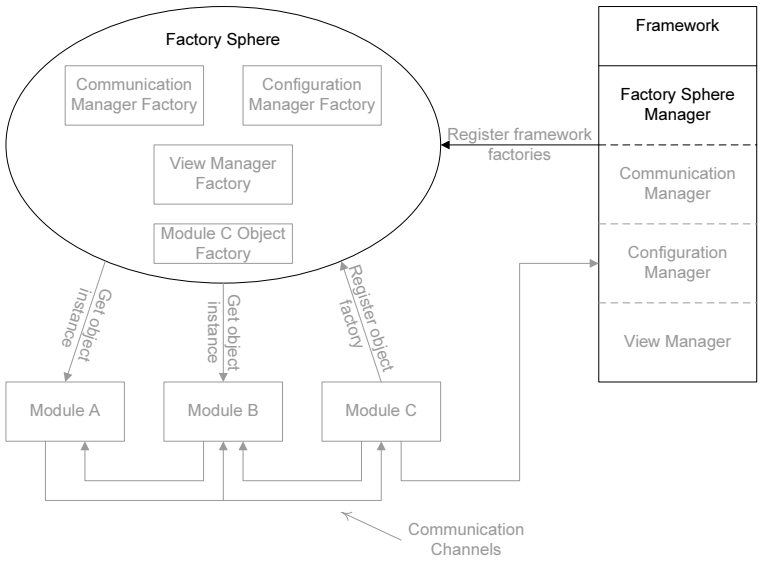

Figure 2.5: Architecture overview for factory sphere system.

software modules have access to the sphere and they can register factories and access factory objects to get object instances.

Inspiration for this system comes from ref. [1], [5] and [8].

2.3.2 Abstract factory

It is not unusual to use abstract factory patterns when designing large software systems since this decrease the bindings between the used objects. This design is especially used when object needs to be shared globally and in reconfigurable system where underlying layers can be substituted.

Figure 2.6 shows the normal design of an abstract factory and is based on the design in ref [1]. Here an interface is used to describe the basic functionality and two subclasses implementing the interface. This design allows substitution of "ModuleA" with "ModuleB", giving the system the ability to use different functionality depending of the situation. The design does however not offer a unified way to control availability of an object, as an example what to do if "ObjectA' is missing in "ModuleA". It can also get very complex and difficult to maintain when there are a large number of objects in the system.

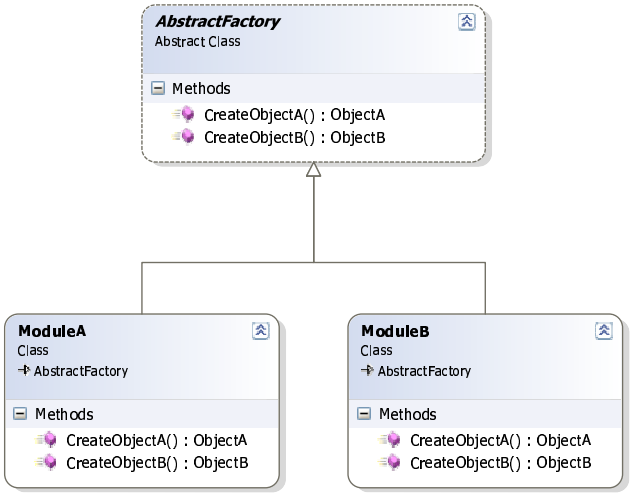

Figure 2.6: Abstract factory design

2.3.2.1 Design qualities

- Binds between objects are weak, allowing easy substitution.
- Forces the developers to make objects that are easier to reuse.
- Simple design.

2.3.2.2 Design drawbacks

- Increased complexity for every object added to the factory system.
- It can be difficult to handle availability of objects.

2.3.2.3 Requirements for new factory system

1. The new factory system for the framework should keep the qualities from the abstract factory system.
2. The new factory system for the framework must be able to handle dynamically adding of factory objects.
3. The complexity of the factory system should not be increased when adding new factory objects to the factory sphere.

2.3.3 Prototype design

This section describes an alternative method for managing factory objects. The alternative factory system is seen as a sphere containing factory objects responsible for creating or sharing object instances. The factory system is thus called factory sphere. The purpose of this prototype is to see if it is possible to create a new type of factory system where the qualities from the abstract factory system is kept and where factory objects can be added runtime.

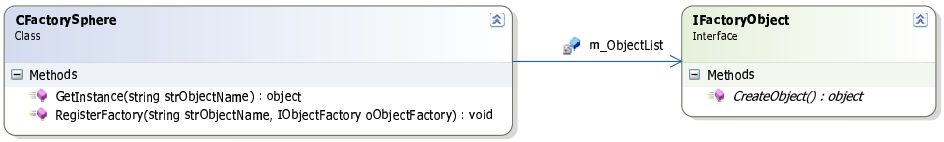

Figure 2.7: Prototype design for factory sphere system.

Figure 2.7 shows proposed design for the new factory sphere system. It contains 2 objects the "CFactorySphere" and "IFactoryObject". The factory sphere object is responsible for containing an arbitrary number of factory objects. The factory sphere handles all the communication between the system and the factory objects.

2.3.3.1 Register factory object

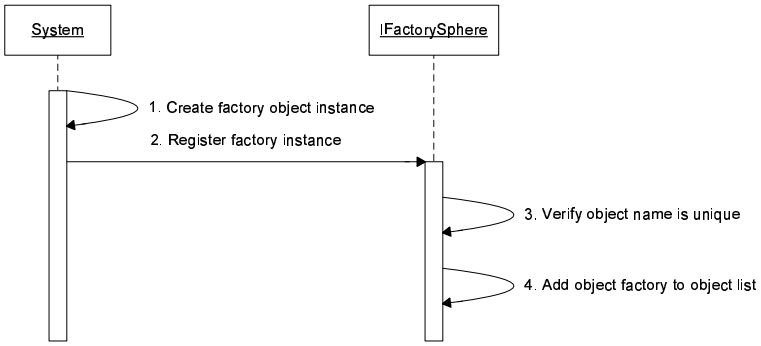

Figure 2.8: Sequence for register factory object.

Figure 2.8 shows how factory objects are registered into the sphere.

1. The system that wants to share an object starts by creating a factory object that can create instances of the desired object.

2. The factory object can then be registered by calling the "RegisterFactory" function using a string to identify the factory object and a reference to the factory object.

3. Before a factory object is registered into the sphere, the factory system verifies the identification for the factory object is unique.

4. If no other objects are registered under the desired name the factory object is then stored in the sphere.

Factory objects are registered into the system while the application is running. This means factories can be registered at any time. However the registration procedure should be contained to the initialization process of the system. The reason for this is to minimize the dependencies between modules at start up. If the factories have been registered during the initialization then all software module have access to the factory when they are started.

2.3.3.2 Get object instance

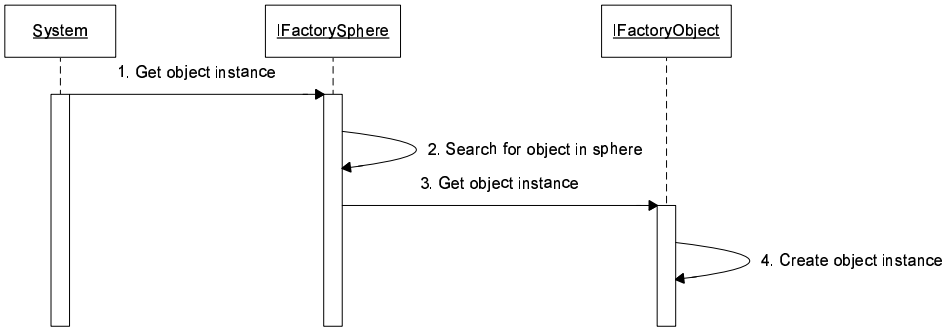

Figure 2.9: Sequence for getting object instance.

Figure 2.9 shows how objects instances should be created.

1. The sequence is started when the system request an object instance.

2. The factory sphere starts searching for the requested factory object using a supplied Id for the factory object. In the prototype design, factory objects are identified using strings.

3. Once a factory object is located the factory sphere system calls the factory objects "CreateObject" function to get an object instance.

4. The factory objects can then, depending of the rules implemented, create a new object instance and return it to the system.

The factory objects in this design are similar to the create functions in the abstract factory design. This object keeps the weak bindings between the objects as required. The prototype design shows the factory object as an interface. This allows for different implementations of the factory objects. These implementations could be different creational patterns like singleton or session patterns.

2.3.3.3 Design qualities

- Binds between objects are weak, allowing easy substitution.
- Forces the developers to make objects that are easier to reuse.
- Can reconfigure the factory system at runtime.
- No increase in complexity when adding a new factory objects.

2.3.3.4 Design drawbacks

- Factory object can be removed runtime.
- Non-specific object instances returned.
- Performance.

2.3.3.5 Prototype summery

The prototype design fulfills the requirements. The register function allows for dynamic adding of factory objects. The create function can create object instances without any increase in complexity. The qualities from the abstract factory have been kept. The design is however not as simple as the abstract factory.

The drawbacks from the abstract factory have been solved in this design. However a new set of problems has been created.

Removal of factory objects

As shortly mentioned factory objects are added runtime. Adding a remove

function to the system could allow for reconfiguration at runtime, removing one factory and replacing it with a new one. This is however not desirable because it would be difficult to keep consistency in the system.

An example could be the system creates 2 instances of specific objects using the Id "TestObject". The factory that is linked to "TestObjects" is now exchanged with another factory object. Now if the system creates instance using "TestObjects" it would get a different object than the 2 first ones. This could lead to unpredictable results because there is not consistency between the objects. The system could also completely remove a factory from the system. This would lead to the question, are the instances still valid. For these reasons removal of factory objects should not be allowed.

However if the consistency problem and validity problems could be controlled, then this could become a strong tool in a modular system.

Performance

The system stored the factory object and identification of these in some type of list. The prototype uses a simple dictionary to store the factory objects in and strings to identify factory objects. Using strings to identify factory objects are not necessarily the best way to go. If the list used contains a large amount of strings it could become a problem.

Object type

The biggest problem in the prototype design is the fact that objects created by the factory objects, are returned without a type making them non-specific. This makes the returned objects useless and a solution must be found to solve this problem.

2.3.4 Prototype front-end design

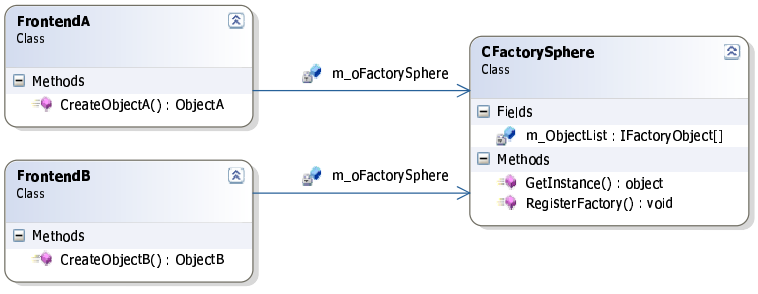

Figure 2.10: Frontend design for factory sphere system.

One way to solve the problem of the non-specific object generated by the factory objects, described in the prototype design, could be by using some of the abstract factory design. This could be creation of front-ends to the factory sphere as shown on figure 2.10. The front-ends contains like the abstract factory a create function. Only responsibility is to get an instance from the factory sphere and type convert it.

This gives a global check from the factory sphere, on availability and the front end gives a type specific object.

2.3.4.1 Design qualities

- Validation on object type.
- Returning type specific objects.

2.3.4.2 Design drawbacks

- Gives more complexity in design.

2.3.4.3 Front end summery

The design gets a little more complex since a create function for each factory object is needed. It is however easier to add and remove factory objects from the front end because of the underlying factory system. The front end do not need the create functions for all the objects and it is possible to use more than one front end on top of the factory sphere.

2.3.5 Requirements for factory sphere

The ideas and experience gained during implementation of the prototypes was used to generate the following requirements, described in this section.

2.3.5.1 Factory sphere interface

FAC1	The interface must have a ”RegisterFactory” function.
FAC2	The interface must have a ”GetInstance” function.
FAC3	The interface must have a ”ReleaseInstance” function.
FAC4	Factories are identified using strings.
FAC5	The factory sphere system may not remove registered objects.

2.3.5.2 Factory object interface

FAC6	The interface must have a ”GetInstance” function.
FAC7	The interface must have a ”ReleaseInstace” function.

2.3.5.3 Factory sphere object

FAC8	This object must implement ”IFacatorySphere” interface.

2.3.5.4 Factory objects

FAC9	These objects must implement ”IFactoryObject” interface.
FAC10	A generic factory object must be implemented.
FAC11	A generic singleton pattern factory must be implemented.
FAC12	A generic session pattern factory must be implemented.
FAC13	A generic pool pattern factory must be implemented.
FAC14	A generic prototype pattern factory must be implemented.

2.3.5.5 Implementation priority

Implementation of the factory sphere system has been prioritized in order to get an overview of where focus should be put. There are 3 levels of priorities these are:

High
: Items that must be implemented before the system works.

Med
Items that would be helpful to have, but not necessary for the overall functionality.

Low
Items that is not necessary but nice to have.

Below is a table showing the implementation priority.

Factory sphere	High
Factory object	High
Generic factory	High
Singleton Pattern	Med
Session Pattern	Med
Pool pattern	Low
Prototype pattern	Low

2.3.6 Design

This chapter describes the final design of the factory sphere manager. This design is based on the requirements described and the experience gained during the implementations of the prototypes.

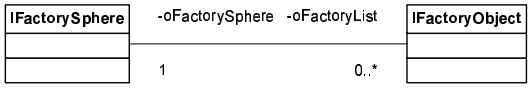

Figure 2.11: Relations between factory sphere objects.

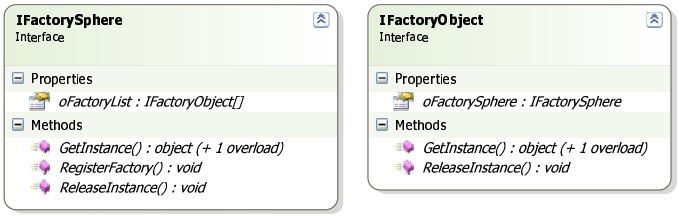

Figure 2.12: Interfaces for factory sphere system.

Figure 2.11 show the relations between the objects and figure 2.12 shows the final interface design.

There are 2 interfaces specified to allow access to the factory sphere. The "IFactorySphere" interface is the main interface for the factory sphere system. This interface is responsible for registration of factory objects and instance creation of objects. In also have a function for releasing objects. This function was added to support certain types of creational patterns, like the session/pool pattern where an object instaces must be released before another instance can be created.

The "IFactoryObject" is responsible for creating instances of a specific object. The factory object implements the creational rules and is responsible for creating the specific object instances.

Once a factory object is registered into the factory system, is cannot be removed again. This means factory object registered must live until this application is closed. This restriction has been added to avoid the problems that could arise if a factory object was to be removed from the system.

2.3.6.1 IFactorySphere functions

GetInstance
: This function returns an object instance. This function takes a string containing the name of the desired factory object.

RegisterFactory
: This function registers a factory object into the factory sphere under a specific Id.

ReleaseInstance
: This function is used to release instances from certain types of design patterns. The function requires the Id of the factory object and a reference to the instance.

2.3.6.2 IFactoryObjects

GetInstance
: This function creates an instance of an object. This function can implement different creational rules.

ReleaseInstance
: This function can be used to release an object instance from a pattern. If the factory object does not implement a pattern that needs a instance releasing, this call can be ignored. The function requires a reference to the instance.

2.3.7 Generic factory objects

In .NET 2.0 it is allowed to create template classes also called generic classes. This can be used to create generic factory objects.

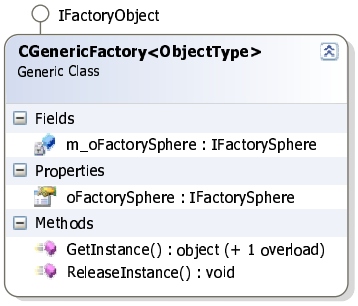

Figure 2.13: Generic factory object.

Figure 2.13 shows how such a class can be created. By implementing the interface "IFactoryObject" into a generic class, the non-generic part of the class is exposed. This allows the generic object to be inserted directly into the factory sphere. To allow a generic class to create instances of an object some restrictions have to be added to the types allowed. The first restriction is the type must implement the "new()" operator. This means all primitive times are excluded from using the generic factory. The new operator can however only be used without parameters meaning the instance created cannot be initialized through the constructor. This means an interface for initialization is required. This leads to the second restriction. In order to initialize the objects they must implement an interface containing functionality for initialization.

2.3.8 Object initialization

As described in the previous section 2.3.7 it is not possible to use parameters in constructors using the described structure.

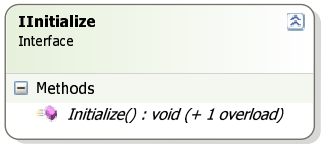

Figure 2.14: Interface for object initialization.

Allowing the object to implement an interface containing functions to initialize the object can solve this. Figure 2.14 shows an interface that solves the problem. During the creational process a check on the object could be done to se if it contains the interface. If the object has the interface, then the initialization functions could be called. These functions can have any parameters required. The function should be called right after the object has been created.

2.3.9 Creational patterns

The purpose of this chapter is to give a short description of the designed creational patterns used. A complete description of the creational patterns can be found in ref [1].

2.3.9.1 Singleton pattern

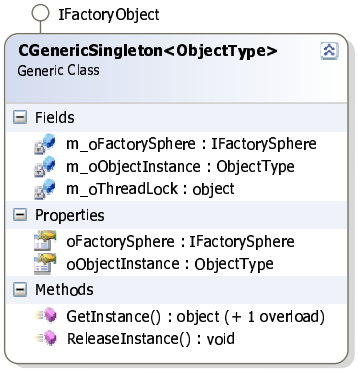

Figure 2.15: Generic singleton pattern implementation.

"CGenericSingleton" in figure 2.15 implements a generic version of the singleton pattern, allowing only one instance of an object to be created. The class implements the "IAbstractFactory" interface to make it compatible to the factory sphere system. The current implementation ignores release calls so once an object instance is created the instance will live until the application is closed.

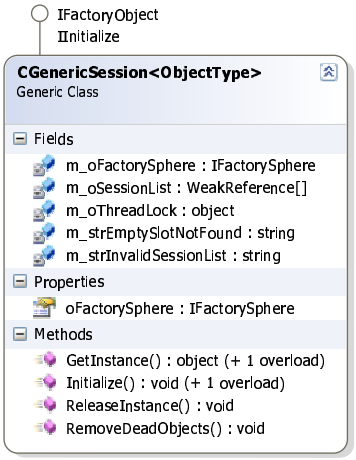

Figure 2.16: Generic session pattern implementation.

2.3.9.2 Session pattern

"CGenericSession" in figure 2.16 implements a generic session pattern, allowing a limited number of instances to be created. The class implements the "IAbstractFactory" interface to make it compatible to the factory sphere system. The object uses "WeakReference" object to create a weak reference to the returned objects. This ensures the garbage collector still releases objects thrown away from the receiving party. This makes the release instance function optional. However since the system wont directly control the garbage collector it is however preferable to release instances in the pattern.

2.3.9.3 Generic front-end object

Like the factory objects it is possible to make a generic object that can function as the front end for an object in the factory sphere.

Figure 2.17 shows how such an object could look like. When the generic front-end is created, it is given the Id of the object to create and reference the factory sphere it has to use. The generic object has also given the type it has to convert to. The create function then gets the object instance through the factory sphere and converts it before returning it. The create function should verify the returned object can be converted to the valid type.

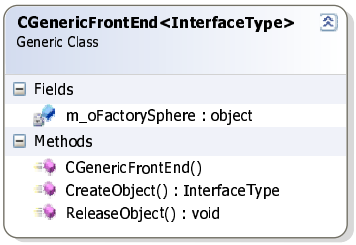

Figure 2.17: Generic implementaion implementation.

2.3.10 Object overview

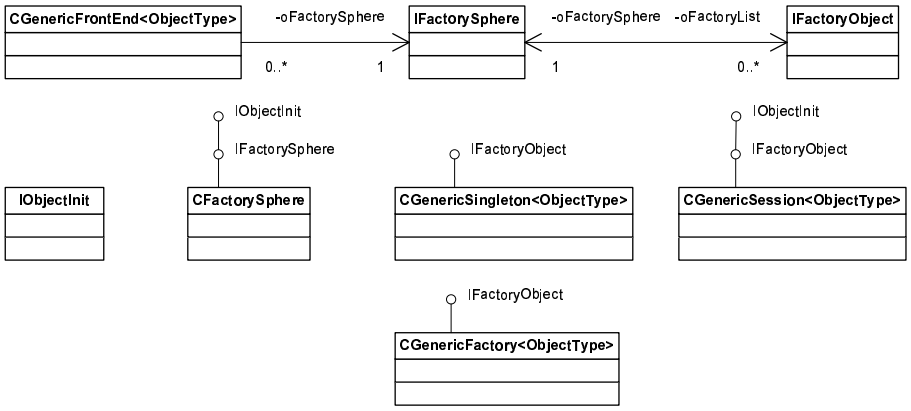

Figure 2.18: Class diagram for factory sphere system.

Figure 2.18 give an overview of the objects and interfaces designed for the factory sphere system.

CGenericSingleton, CGenericSession, CGenericFactory are all implementations the IFactoryObject.

CFactorySphere are an implementation of the IFactorySphere.

2.3.11 Test

The factory sphere system works as expected. To test system a small unit test application was created. The test protocol used, can be found in appendix G.3

2.3.12 Summery

It is surprisingly easy to use the generic factory object, register and get object instances from the factory system.

The factory sphere system can be used on any type of objects since factory object that create the objects can be customized if needed. The generic factory objects designed can in most cases be used without any modifications.

The factory sphere has the same qualities as the abstract factory design but with the addition that it can handle new object added at runtime.

The downside to the factory sphere could be performance depending on how factory objects in the factory sphere are searched and identified. This is something that must be looked at if this is to be used in the future. The factory sphere also contains more components than the original abstract factory making it more complex to implement.

The qualities of the factory sphere system do however make it attractive to use. The system can be reconfigured without the need of recompilation of any parts of the system.

Only 3 of the required factory objects have been implemented, these are Generic factory, Singleton factory and Session factory.

2.4 Module communication

The purpose of this section is to give a description on how events/communication can be performed between modules.

The section starts with a small description of where the communication system fits in to the framework architecture:

1. A small analysis was done to get an overview of the desired communication architecture.
2. A description of the mediator pattern is given, listing qualities and drawbacks. A small description is given on how this mediator pattern helped to inspire the design.
3. A list of requirements is given to specify the communication systems functionality.
4. Interfaces for the communication system are designed, describing how call through the communication system can be done both synchronously and asynchronously, qualities and drawbacks. The interface design ends with a small summery.
5. A small description of how to manage channels and protocol used are given.
6. A small test is described and used to test the implemented communication system.
7. A summery with drawbacks and qualities of the design is given.

2.4.1 Architecture overview for Communication system

The highlighted components in figure 2.19 shows the items for the communication system to be designed. The communication manager is a part of the framework and is registered as a generic singleton factory when it is initialized. This is done to give all other software module access to the same communication channel in the system. The communication manager is as the factory system one of the main components in the framework. Its purposes are to give all software modules a unified way of communicating with each other. A unified way of communication should minimize the bindings between all software modules.

Concept 5 in section 1.4.5.5 is the main inspiration for this design. But also ref. [1] and [3] have been used.

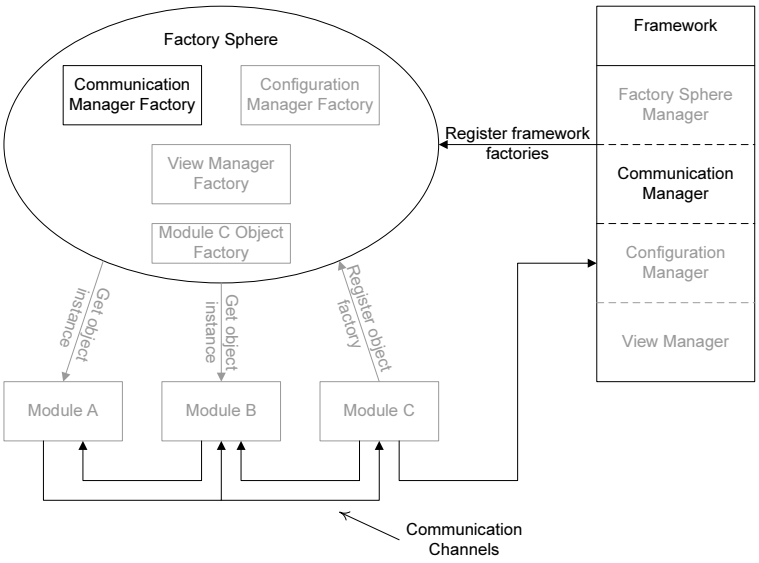

Figure 2.19: Architecture overview for communication system.

2.4.2 Analysis

The software modules in this system do not have a direct way of communication since this is not supported in the interface. Instead I have borrowed an idea from network theory to create communication between the software modules.

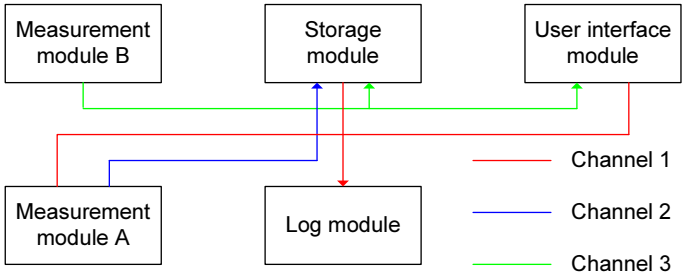

Figure 2.20: Concept for module communication.

Instead of creating interfaces and start sharing object through the factory system, this idea is based on using communication channels. Figure 2.20 shows the concept of how this could work. The figure displays 3 channels. Channel 1 is a channel used by the user interface, module A and the storage module to send events to a log module. Channel 2 is used by module A to send measure-

ment data to the storage module. Channel 3 is used to send measurements from module B to the storage system and user interface.

A communication channel is a place where modules can register listeners to receive data. If multiple listeners are registered on a channel (Channel 3) then data is broadcasted to all listeners. A channel can also have multiple senders (Channel 1) allowing modules to share functionality in one module.

2.4.3 Mediator pattern

The communication manager that controls the communication channels can be seen as a kind of Mediator pattern. The object contains a list of functions to be called. Each list contains an unique Id. When calling a list using the Id, then all the functions in the list are executed.

The mediator pattern is designed to execute some type of interaction between objects without the objects know anything about the other objects. The mediator pattern is described in ref. [1] page 273.

The mediator pattern can however not directly be used because the mediator pattern have to know all the objects or at least an interface to these objects.

2.4.3.1 Design qualities

The mediator pattern have the following qualities that is attractive to keep:

- Gives weak bindings between objects.
- Give object the ability to interact without knowing each other.

2.4.3.2 Design drawbacks

The mediator does also have some drawbacks. These are:

- Mediator needs to know all the objects that requires interaction.
- Mediator requires a unique implementation for each interaction type.

2.4.3.3 Summery

The design should focus on solving the drawbacks of the mediator pattern and keep the qualities.

2.4.4 Requirements

COM1	The interface must contain a function for adding channels.
COM2	The interface must contain a function for registering a listener on a channel.
COM3	The interface must contain a function for removing a listener on a channel.
COM4	The interface must contain a function verifying the existents of a channel.
COM5	The interface must contain a function for sending data to a channel.
COM6	Channels must be identified using strings.
COM7	Listeners can be registered as synchronized.
COM8	Listeners can be registered as asynchronized.
COM9	Channel may not be removed.

Channels must not be removed from system once the application is running. Removing a channel could lead to unpredictable results. As an example a defect software module is loaded. When the software module is initialized, it removes all channels in the system. This scenario could be devastating for the entire system.

Listeners may however be removed. The reason for this is because in order to remove a listener from a delegate, a reference to the listener is required. This means only the owner module can remove a listener from a channel. If a defect in the module would remove all its listeners it would not be devastating for the entire system. The problem would be contained in the specific software module.

2.4.5 Channel interface

First the focus have to be on how communication should be performed between the modules. To solve the problem of the mediator needing to know all the objects required for interaction, it could be turned around. This means all the software modules that needs interaction should know the mediator. This however requires some interfaces to be specified.

The channel interface should be used to give a unified way for communication

between the modules.

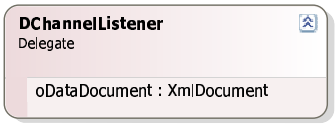

Figure 2.21: Channel listener interface.

The method chosen is shown on figure 2.21. It is a delegate that contains a reference to a XML document.

A delegate can be used to contain a list of functions following the specified format. This means that one delegate can function as one channel simply by sharing the delegate among the modules that needs to interact. The modules can then register functions into this delegate. A module calling this delegate does not need to know any thing about the other modules.

Using delegates in Compact Framework 2.0 does however have one large drawback. This only supports synchronous communication. This means if a module calls the channel it will have to wait until all the functions in the delegate have been executed. This is not always desirable and for this reason an asynchronous way of communication must be designed.

2.4.5.1 Asynchronous communication

In order to create asynchronous communication I have added the following classes.

Figure 2.22 show the objects used to create an asynchronous function call. The "CListener" is an object that is used to hold the references to a listener function that needs to be called. The "CListener" object specifies whether the given listener is called synchronous or asynchronous. When the "CListener" object is registered onto a channel as an asynchronous listener. It is the "CListener" "AsyncListener" listener function that will be registered into the channel. For synchronous listeners it is the "m_oListener" attribute in the "CListener" object that will be registered into the channel.

Figure 2.23 and 2.24 shows the sequences the done when registering and calling an asynchronous class.

The thread created in the "CListener" object is thrown away and left to itself

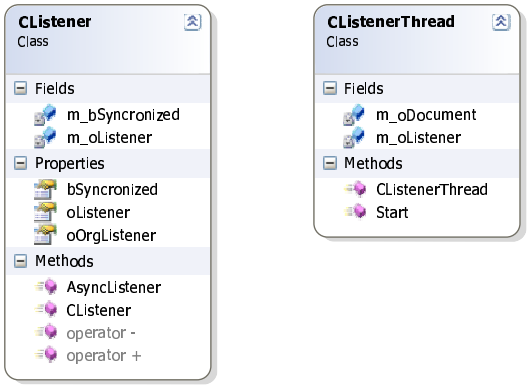

Figure 2.22: Asynchronous listener objects.

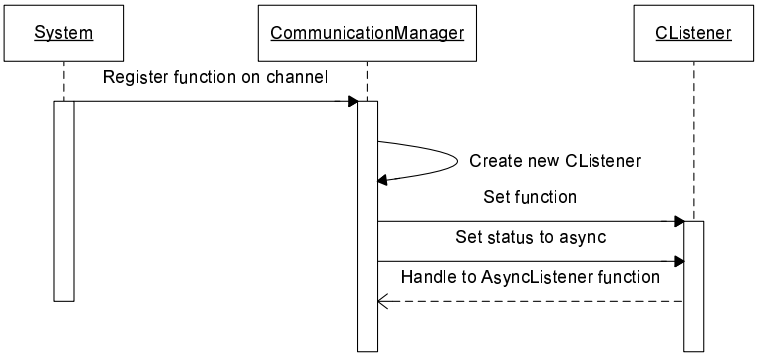

Figure 2.23: Sequence for register asynchronous listener.

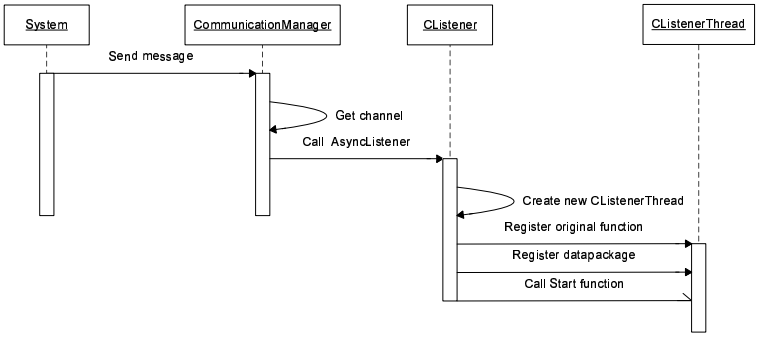

Figure 2.24: Sequence for executing asynchronous listener.

once it has been started. This means no one have a reference to the create thread at it will simply die once the function call is completed. Concerns were raised whether this approached would lead to a memory leak. For this reason a small test application was created to test for a leak. The test can be seen in appendix F.

2.4.5.2 Design qualities

- Unified interface.
- Can add listener's runtime.
- Can execute functions synchronous and asynchronous.
- Gives weak bindings between objects.
- Give object the ability to interact without knowing each other.

2.4.5.3 Design drawback

- Channel needs to be shared between all modules.
- Can only support one type of functionality.
- Cannot support complex interaction between modules.

2.4.5.4 Summery

The current system has kept the qualities from the mediator. It has also solved the drawbacks from the mediator. Giving the system a unified interface for communication and using delegates removes the need for a unique implementation.

However some new drawbacks have also created. Unifying the interaction in a delegate makes it difficult to implement complex interaction between modules. As an example the system cannot get data from one module and then send it to another in the same delegate. However since interaction between modules should be as limited as possible this is not considered a problem.

A way to share a channel between the modules have to be found, since all the modules that need this channel for interaction, needs to be able to either register a function or call the delegate. One way to solve the problem could be to use the designed factory to handle the sharing.

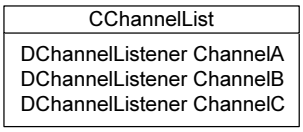

Figure 2.25: Concept of channels list.

One delegate should only be used for one type of functionality. This should give the best design. A function crammed with functionality can be difficult to implement and maintain. For this reason a solution must be found in order for the system to handle more than on channel.

Validation of the data transmitted has to be performed by the receiving modules. Creating a global verification system is not an acceptable solution. Not only could a global verification system function as a bottleneck, it would also very complex since it would have to be able to validate all types of protocols. Validating received data in every software module does however decrease performance.

2.4.6 Managing Channels

Figure 2.25 shows a prototype for an object that could be used to handle more than one channel. This figure contains 3 channels and if it were shared between the modules in figure 2.20 they would all have access to call and registered functions. This object could be registered into the factory system. Since all modules have access to the factory system they could then retrieve it. This would however require an interface to be specified.

2.4.6.1 Communication manager interface

Figure 2.26 shows the designed interface for a communication manager object that is responsible for handling the communication channels.

AddChannel
: This function is responsible for adding a channel to the system. Strings identify channels, hence the function requires a string containing the channel name.

AddListener

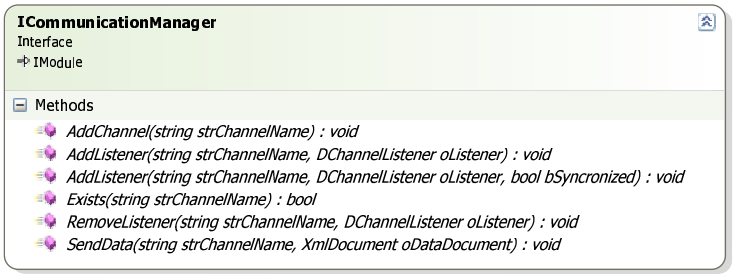

Figure 2.26: Interface for communication manager.

This function is used to add a listener to the channel. The function requires the channels name and a reference to the listener to be registered. By default listeners are added as asynchronous functions. A secondary function is available where the synchronized parameter can be supplied.

RemoveListener
This function removes a listener from a channel. The function requires the channels name and a reference to the listener to be removed.

Exists
This function returns a Boolean to tell if a channel is found in the internal channels list. The function requires a channel name to be supplied.

SendData
This function sends data out on a selected channel. The function requires a channel name to be supplied and a data package.

2.4.6.2 Protocols

The communication system needs some specified protocols in order to give a structured communication system. Since the protocols are XML documents, the documents required and their attributes are specified in appendix E.

2.4.7 Test

To verify the system works a small test application have been made to perform unit testing of the communication system The test protocol used can be found in appendix G.2. The result of the test shows the system works as expected.

2.4.8 Summery

The design of the communication gives a system that is very flexible. An arbitrary number communication channels can be created to support an arbitrary number of functionality. The system uses an XML document as a parameter allowing for an arbitrary number of attributes to be supplied to the functionality

This design has a few drawbacks that have already been discussed. However performance could also be a problem using this system. The XML document itself is not parsed to all the listeners, but only the reference. However all the attributes in the XML document must be converted from string to the needed format and when added to the document they must be converted from the format to string.

Data integrity could also be an issue, since it is only a reference that is passed between the listeners. One listener could modify the data resulting in invalid data for all the other listeners. For this reason listeners should be restricted to read received data only.

The few drawbacks this system has are outweighed by the qualities of the system. It gives us a way to runtime configures how modules should interact without the module has to know any thing about each other, expect channel name and protocol.

The given design contains 2 add functions for adding channels. The only difference between the functions is the parameter "bSyncronized". This parameter is used to select whether the listener registered is synchronous or asynchronous. The add function without the parameter will always register listeners as asynchronous. This means the function containing the parameter most likely only will be used to register synchronous listeners. The next version should therefore remove the this parameter and have add function for registering listeners directly to synchronous or asynchronous.

2.4.8.1 Design qualities

- Unified interface.
- Can add listener's runtime.
- Can remove listeners runtime.
- Can execute functions synchronous and asynchronous.
- Gives weak bindings between objects.

- Give object the ability to interact without knowing each other.
- Flexibility in communication.

2.4.8.2 Design drawbacks

- Channel needs to be shared between all modules.
- Cannot support complex interaction between modules.
- Performance.
- Data integrity.

2.5 Configuration manager

The purpose of this section is to describe how configuration files should be handled.

1. To start with a small description of where the configuration system fits into the framework architecture is given.
2. A small analysis was done to get an overview of how configuration files should look and be controlled.
3. Description of security issues regarding configuration files is given and also a method for integrity testing configuration files.
4. A design of the main configuration files is given, with a description of how external and custom designed configuration files could be added.
5. The section ends with a small summery and a list of qualities and drawbacks of the design configuration system

2.5.1 Architecture overview for Configuration Manager

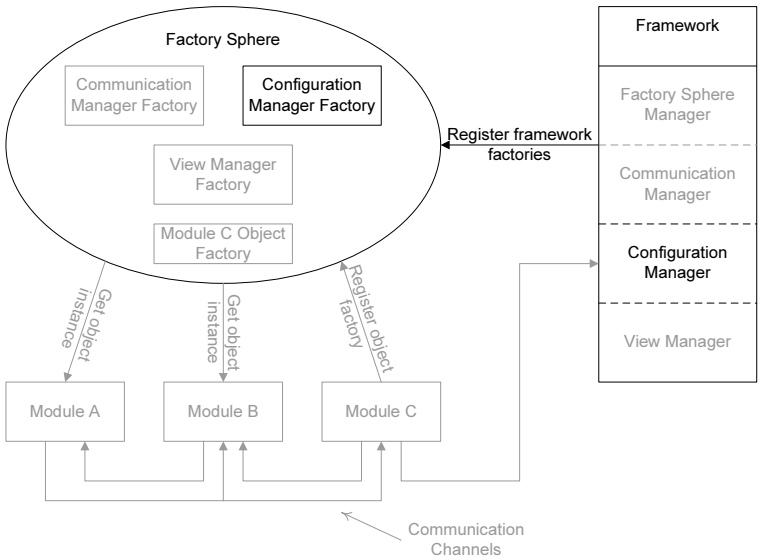

Figure 2.27: Architecture overview for configuration manager

The highlighted components in figure 2.27, shows where the configuration system is located in the framework architecture. The configuration system is a part of the framework and is used to load and save configuration files. The configuration manager registers a factory allowing all modules to access it. It registers the factory object using a generic singleton factory. This is done to increase performance since loaded configuration files only have to be loaded and parsed once.

Inspiration for this module is mainly based of the module described in Concept 2 section 1.4.5.2.

2.5.2 Analysis

The configuration system uses XML files. The reason XML files have been chosen is because they are easy to edit and readers and writers are supported in compact framework. The readers and writers make it easy to extract and modify the configuration from the application.

2.5.3 Security

There are several problems involved when having external configuration files. These are:

Data corruption
: This could result in impassable configuration data or invalid configuration of system. An invalid configuration could harm the patient or result in wrong treatment. Data corruption could come from a damaged disk or by accidentally overwriting the configuration file. External editing could also be a problem. Especially since the files are in clear text.

License restriction
: Depending on the product purchased, it could be necessary to implement certain license restrictions into the configuration files. These restrictions should be protected so the buyer can't gain access to restricted services by editing the configuration files.

These problems can be solved by digitally signing the configuration files when created or modified.

2.5.3.1 Signing using RSA

Signing data using RSA keys are done by following the procedure shown in figure 2.28.

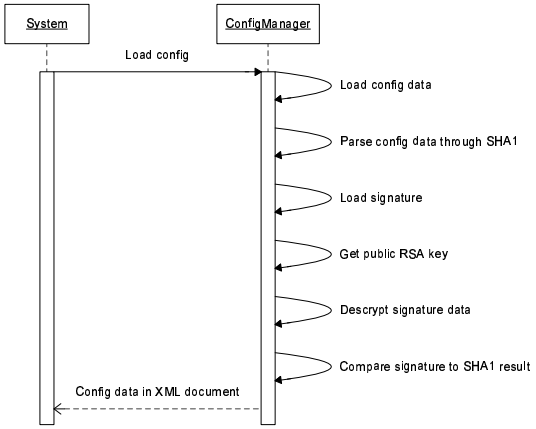

Figure 2.28: Sequence for signing data.

Data are first parsed through a hash algorithm in this case SHA1 ref. [13]. This generates a unique hash of the file. This hash value is then encrypted using the private key from the RSA key pair to prevent editing. This gives us a signature of the file.

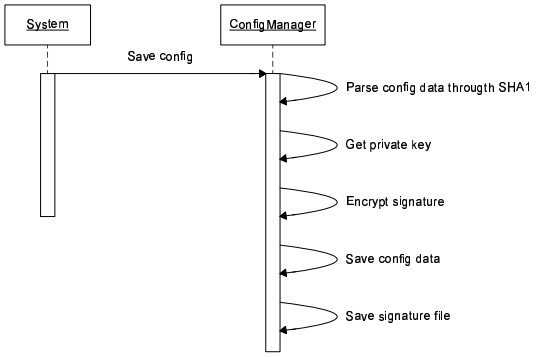

Figure 2.29: Sequence for verifying data.

Verification procedure is shown on figure 2.29. Data is passed through the SHA1 algorithm. If the data have not been changed since it was signed then the hash value is the same as it was when it was signed. To verify this, the public RSA

key is used to decrypt the signature and then compare the two hash values. If the hash values do not mach then the file is declared corrupt.

The strength of the signature depends on the hash algorithm used, the number of hash algorithm used and the RSA key size used.

2.5.4 Requirements

In order to sign the configuration files some requirements must be met by the configuration system.

CFR1	The system must be able to handle static configuration.
CFR2	The system must be able to handle dynamic configuration.
CFR3	Only the public RSA key must be available for static configuration files.
CFR4	Private RSA key must be available for dynamic configuration files.
CFR5	A files signature must be verified when loaded.
CFR6	A files signature must be updated when saving.

2.5.5 Design

The functions to load and save XML files are built into compact framework 2.0. So are RSA functions to sign and verify data. This means the only thing to be designed are the file buildup.

2.5.5.1 Main configuration file

The main configuration file is called "FrameworkConfig.xml". This file is static configuration file containing the overall configuration of the system and links to external configuration files.

The configuration files consist of sections following the format

```
<ModuleConfig ID="">
  <External></External>
  <SaveAllowed></SaveAllowed>
  <ConfigFile></ConfigFile>
```

```
  <ConfigSignatureFile></ConfigSignatureFile>
  <KeyID></KeyID>
  <ConfigData>
  ...
  </ConfigData>
</ModuleConfig>
```

ID

This attribute is part of the "ModuleConfig" section and is used to separate the sections from each other. The ID normally contains the GUID value for a specific module.

External

This attribute is used to signal if the module configuration is located in an external file or if it is embedded in the main configuration file. If the attribute contains the value "True" then the configuration is located in an external file. If the value is "False" then the configuration is embedded in the main configuration file.

SaveAllowed

This attribute is used to signal if the configuration is static or dynamic. If the attribute contains the value "True" then it is allowed to modify and save the file. If the value is "False" then modification of the configuration file is not allowed.

Restrictions

This value must be set to "False" if the "External" attribute is set to "False". This is because the main configuration is static and cannot be changed.

ConfigFile

This attribute contains the path and filename for an external configuration file to load or save.

Restrictions

This attribute requires the attribute "External" to be set to "True". Else it will be ignored

ConfigSignatureFile

This attribute contains the path and filename for a file containing the signature of the external configuration.

The signature file is a primitive XML file containing a "Signature" tag that holds a hex encoded signature.

```
<Signature>83,6F,4C,... ...,5D,85,78</Signature>
```

KeyID
This attribute is used to point to a RSA key in the key manager.

Restrictions
This attribute requires the attribute "External" to be set to "True". Else it will be ignored

If the attribute "SaveAllowed" contains the value "True" the "KeyID" attribute must point at a private RSA key

ConfigData
This attribute is used to encapsulate configuration data for a module if the attribute "External" is set to "False". Else this attribute is ignored.

2.5.6 Summery

There are performance issues that have to be considered in the final system. Every time a configuration file is loaded it has to be parsed through the signing algorithm and then parsed by the XML system. Saving a configuration also requires a parse through the signing algorithm. This is time consuming could give problems if the selected processor have a low execution capacity.

The design is relative simple because of the XML usage. XML files are well supported by the compact framework giving a lot of tools to manipulate these files. The compact framework also supports the signing system. This means most of the design for the configuration system is in how the files should be built.

2.5.6.1 Design qualities

- The design gives the ability to control who is allowed to edit files.
- The design gives a good integrity test of configuration files.
- The design gives "easy to edit" configuration files.
- The design uses a well supported system for modifying configurations.

2.5.6.2 Design drawbacks

- Performance.

2.6 Controlling user interface

The purpose of this section is to describe how user interfaces should be controlled.

1. To start with a small description of where the control system fits into the framework architecture is given.
2. A small analysis of control systems is given, describing how an application controller pattern could be used. The analysis list identified qualities and drawbacks for this pattern.
3. A list of requirements are given to describe the view systems functionality.
4. A design of the control systems is given. This design contains descriptions of the designed interfaces, object relations, interaction between objects, qualities and drawbacks of the design and a small summery.
5. A description of how the control system should be configured is also given. The section describes how configuration files should be built and discusses how to move from state diagram to a configuration file and bindings between views. It ends up with listing qualities and drawbacks for the methods and gives a small summery.

2.6.1 Architecture overview for View Manager

The highlighted components in figure 2.30, shows where the view manager is located in the framework architecture. The view manager is used to link user interface views together. A view is one screen in the user interface. The purpose of the view manager is to allow several views to be easily configures into a coherent user interface. The system allows easy reuse and exchange of view in the user interface.

The main inspiration for this design was the Application Controller pattern described in ref. [2].

2.6.2 Application Controller

The "Application Controller" [2] is a design pattern used in enterprise applications to control interaction between user interfaces and execution of commands

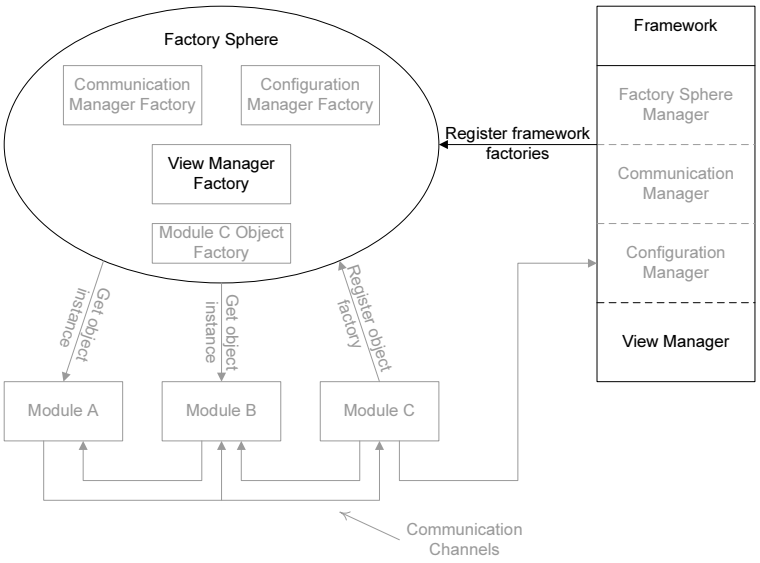

Figure 2.30: Architecture overview for view manager.

based on system and user input. Figure 2.31 show a sequence diagram for an application controller. This design is taken from ref [2] page 379.

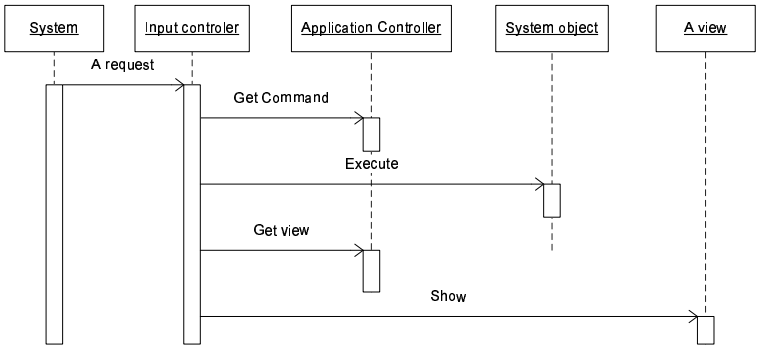

Figure 2.31: Application Controller pattern sequence.

The "Application Controller" pattern consist of several patterns that interacts to give this functionality. Some of these are state patterns used to control the application state and knowledge objects that contain command sequence, list of user interfaces for specific application states. Many of these patterns make it possible to externally configure the system, exchange functionality and user

interfaces as needed. This gives an abstraction between the user interface and underlying business logic, allowing for substitution of modules in both layers.

2.6.2.1 Design qualities

- Easy substitution of user interface.
- Easy substitution of business logic.
- Easy reuse of user interfaces.
- Easy reuse of business functionality.
- Give an overview of user interaction at design time.

2.6.2.2 Design drawback

- Performance.
- Requires large amount of design documentation.
- Design is for large systems.

2.6.2.3 Summery

The "Application Controller" pattern can be complex to use, and since the target application is an embedded system. One way to use this application is to reduce the pattern and only take a part of it. The part that is interesting to take is the part that controls the user interface. The part that controls function execution should be ignored. This can be ignored since interaction between the modules should be done through the communication channels. Also the main purpose of the user interface is to present data send from the underlying system, minimizing the amount interaction with the underlying system.

Performance could be an issue depending of how views are identified in the knowledge object. The design performance will not be considered because it is to demonstrate functionality only. For easy implementation the identification will de done using strings.

The easy reuse and substitution of user interfaces should be kept in the final design

2.6.3 Requirements

VR1	View must be signaled when opened.
VR2	View must be signaled when closed.
VR3	View must be able to access input controller.
VR4	Input controller must be configured externally.
VR5	Views should know where the system comes from.
VR6	Views should know where the system is going.

2.6.4 View system design

Using the "Application Controller" pattern as inspiration. The following system have been designed.

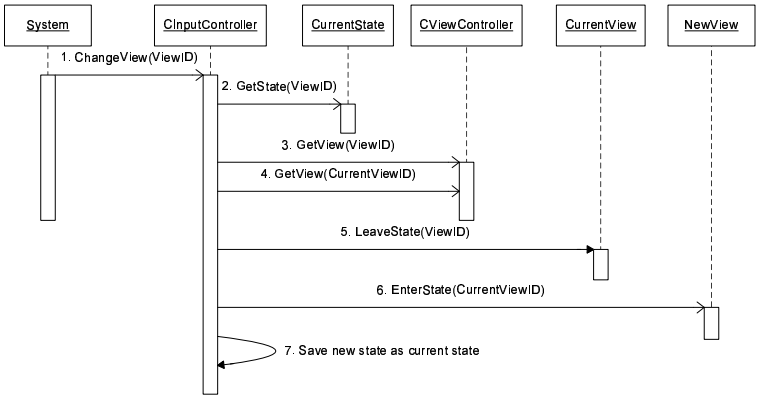

Figure 2.32: Sequence for changing views.

Figure 2.32 shows the sequence used to change views in the "View Controller".

1. The system starts by sending a request to the input controller to show a specific view.
2. The input controller asks the knowledge object "CurrentState" to return the state containing the requested view.
3. When a valid state is returned, it is possible to ask the knowledge object "CViewController" for a reference to the physical view.
4. For bookkeeping a reference to the current view is also needed.

5. Using the reference to the current view it is now possible to tell the view that the system is leaving it (VR2, VR6). When leaving a view the input controller gives an Id to the view the system is going to.
6. Once the system have left a view it can signal the new view that it is active (VR1, VR5). When signaling the view and Id is supplied describing where the system comes from.
7. The last this is to update the current state attribute. Storing the new state in the current state attribute does this.

The views are responsible for showing and hiding themselves when entering and leaving.

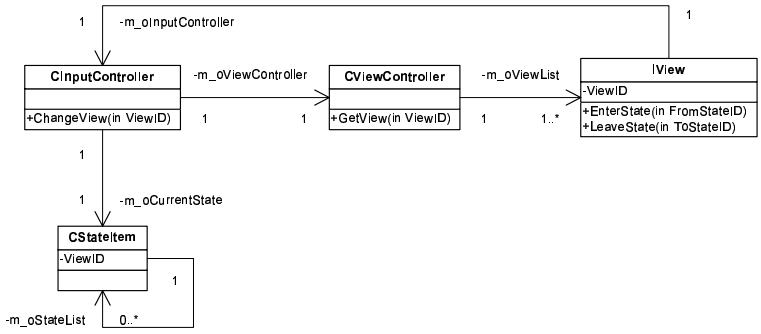

Figure 2.33: Class interaction for view system.

Figure 2.33 shows a class diagram for the "View controller".

CInputController

The input controller object is a part of the state pattern used to control the current active view. This object is responsible for storing the current state, validate input and change the system state depending on input given and the current state. It is also used to signal the views when the system enters or leaves a view.

CStateItem

This object contains information about a specific state. The object itself is a knowledge object. It contains an Id for a specific view to be shown and information's about the states the view can go to.

CViewController

This is a knowledge object combining view identification to a physical reference of a view.

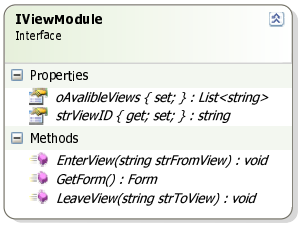

Figure 2.34: IViewModule interface.

IView
This is an interface used to access functionality in the views.

2.6.4.1 View module design

Figure 2.33 show the "IView" as a component needed by view controller system. The interface designed is shown in figure 2.34.

oAvalibleViews
This property is to give the view a list of view Id's this view is allowed to go to.

strViewID
This property is to get of set the view Id for the view.

GetForm
This is a function used by the view system to get access to the "Form" object to show the user interface.

EnterView
This function is used to signal a view that it is now active.

LeaveView
This function is used to signal a view that it is now inactive.

2.6.4.2 Design qualities

- Easy reuse of views.
- Easy substitution of view.
- Easy reconfiguration of views.

- Give an overview of user interaction at design time.

2.6.4.3 Design drawbacks

- Limited interaction between views.
- Performance.

2.6.4.4 Summery

The view system design is a reduced version of the "Application controller" pattern. This design only controls the user interface part. This results in a lighter and easier system that can be implemented into an embedded system. It does however create a new drawback in the design. Since the control system only controls the views and not function calls, this limits the interaction between the views significantly. The views only get information's about the views available, where the view comes from when entering and where it is going when leaving. However the few drawbacks there are in this system do not outweigh the qualities in the system. The ability to substitute a view with another just by changing the ID makes the system easy to adapt and gives an easy way to reuse user interface.

2.6.5 View configuration

As described the input controller shown on figure 2.33 is just a state machine that based on some input is capable of changing state. To make the state machine as dynamic as possible is should be build up around a configuration file describing the states.

2.6.5.1 Configuration file

The configuration files have two main sections. The fist is a list of views to be loaded and the second it a list that describes each state.

Before generating a configuration files a state machine should be drawn to get an overview of the states in the system.

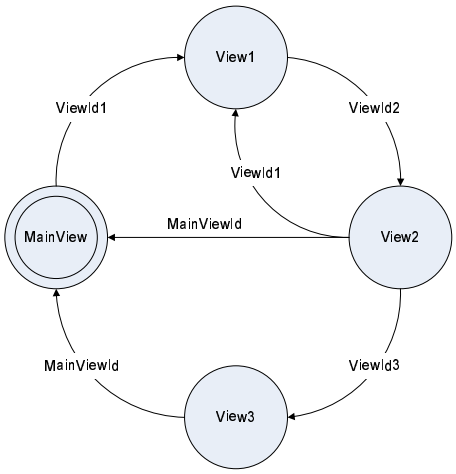

Figure 2.35: State diagram for views.

Figure 2.35 show the system this section will use to show the configurations. This system has 4 views. From this the first part of the configuration can be defined.

The view list is build as following

```
<Views>
  <View>
    <ViewID>MainView</ViewID>
    <ModuleFile>Dll filename for the view</ModuleFile>
    <ModuleObject>Object name of the view</ModuleObject>
  </View>
  <View>
    <ViewID>View1</ViewID>
    <ModuleFile>Dll filename for the view</ModuleFile>
    <ModuleObject>Object name of the view</ModuleObject>
  </View>
  <View>
    <ViewID>View2</ViewID>
    <ModuleFile>Dll filename for the view</ModuleFile>
    <ModuleObject>Object name of the view</ModuleObject>
  </View>
  <View>
    <ViewID>View3</ViewID>
    <ModuleFile>Dll filename for the view</ModuleFile>
```

```
    <ModuleObject>Object name of the view</ModuleObject>
  </View>
</Views>
```

The section "View" is an entry that describes one single view.

ViewID
This attribute is used to give a view a unique Id to identify it by.

ModuleFile
This attribute contains the filename and path for a DLL file containing the view.

ModuleObject
This attribute contains the name of an object in a DLL file that functions as the entry point for a view module.

The combination of the filename and object name uniqly described the module to be loaded. This means more than one view can be located in the same DLL file.

The second part of the configuration file describes the relations between the views. From figure 2.35 the following can be described:

```
<States>
  <InitialView>MainViewId</InitialView>
  <view>
    <ViewID>MainViewId</ViewID>
    <State>ViewId1</State>
  </View>
  <View>
    <ViewID>ViewId1</ViewID>
    <State>ViewId2</State>
  </View>
  <View>
    <ViewID>ViewId2</ViewID>
    <State>MainViewId</State>
    <State>ViewId1</State>
    <State>ViewId3</State>
  </View>
  <View>
    <ViewID>ViewId3</ViewID>
```

```
    <State>MainViewId</State>
  </View>
</States>
```

InitialView
: This attribute defined the view to be showed at startup. In figure 2.35 this is showed as the initial state.

The configuration files contains a list of "View" sections. These sections describe the state a specific view is allowed to go to by listing the Id's for the allowed states.

ViewID
: This attribute is used to define the view for the state list.

State
: This attribute contains the Id of a view where the defined ViewID is allowed to go to. The section can contain 1 to many of this attribute.

As an example of how a section to build up it can be seen "View2" can go to "MainView", "View1" and "View3". The section that defined the states for "View2" contains the Id for these 3 views.

2.6.5.2 Dependencies between views and input controller

Defining the state machine is however not enough to make it work. The views have to give feedback to the input controller in order to change views.

The input controller required the viewId's to be given as input when changing views. This means button or actions in a view have to be linked to a ViewId.

This can be done using a simple configuration file like this.

```
<ButtonLinks>
  <ButtonID>ButtonA</ButtonID>
  <ViewID>ViewId1</ViewID>
</ButtonLinks>
```

This creates a dependency between the views and the input controller because the view needs to know what view it has to go to. This link is unavoidable,

however keeping it in configuration files give the possibility to reconfigure the views without recompiling the views.

The dependency could be moved to the view controller by modifying it to respond to button Id's instead of ViewId. This however would require the input controller to know about the buttons in the views. Where this dependency should be considered before a final system is created.

2.6.5.3 Design qualities

- Easy to reconfigure view interaction without recompiling views.

2.6.5.4 Design drawbacks

- There are dependencies between the input controller and views.

2.6.5.5 Summery

The way the view system is build up generates some dependencies between the views and the input controller. A larger analysis should be performed to see if this dependency could be reduced and where it should be located. It will probably not be possible to remove this dependency because views can have an arbitrary number of buttons and can go to any view. This means some one have to know about the internals of the view and how the view should navigate to other views.

2.7 Design summery

This chapter describes how modules should be designed and describes 4 implemented software modules.

2.7.1 Module design

The design shows how a basic software module should look. Access to software modules can only be done through a simple interface. This limits bindings be-

tween software modules. All software modules rely on external help for all types of communication between software modules. This has been done to remove any slippage problems when it comes to communication between software modules.

Slippage problems often shows themselves when module that have not been designed to communication with each other suddenly have to. The interfaces between these are in most cases not compatible. Forcing modules to communicate through a common interface removed the problem.

2.7.2 Factory sphere

This module was developed to give a common way for all software modules to share factory objects. The system works by letting software modules register a factory object into the factory sphere, which is responsible for creating object instances. The factory sphere consists of a number of general interfaces designed to allow almost all types of objects to be created. Modules should also be able to register any type of factory objects into the sphere. This generality have been designed into the module to allow easy evolution of the entire framework. The only restrictions in the factory sphere are the factory objects have to implement a simple interface. If modules exclusively rely on the factory sphere for communication with other modules, slippage problems could become a problem.

The factory system used the Abstract Factory patterns as baseline of comparisons to develop a system where factories can be added runtime. The final result has almost the same qualities as the original factory. The biggest drawback is this system is more complex to implement.

The factory sphere implements a number of generic creational patterns that can be used to create object with. These are session and singleton patterns. The generality of the factory sphere could be seen as a sign, that the factory system could be a design pattern. However before this can be determined analysis of this must be done.

2.7.3 Communication Manager

This module controls communication channels between modules. The communication manager have been developed to minimize the effect evolution can have on communication between modules and thus reduce slippage problems. The communication manager lets modules create channels to communicate over using parsed XML documents as protocols. Not only gives this method a very

flexible method for communication, but also a unified interface for this.

The communication system is relative simple to implement and use. But it has some problems. Data have to be packed into and out of a XML document. This requires time and lower performance. This means this system has to be tested on the final hardware platform before any conclusion of usability can be done. There is also a problem regarding data integrity. To increase performance in the communication system, it is only the reference to the XML document that is parsed to each listener. This means if one listener modifies the protocol contents it could affect other listeners.

2.7.4 Configuration Manager

This module can be used by other modules to load and save configuration files and test integrity. The module is relative simple to implement because the configuration files are XML documents and the .NET framework has XML classes capable of this. Most of the work in this module was on the implementation of the integrity test.

2.7.5 Controlling user interfaces

To control the views in the user interface a reduced version of the application controller pattern was implemented. The implemented solution works, but the overall result for this software module is not a complete success. Reducing patterns cannot be recommended. It is time consuming and the end result is not as usable as could be desired. The implemented view manager has some unresolved issues regarding how dependencies between the views should be. There is also a lot of work to be done regarding configuration of the system. Before this software module should be used a cost/benefit analysis of the software module should be done and compared to a normal implementation of a user interface.

If this module was to be redesigned it could be more beneficial to use ref [5] and combine patterns into a useful solution.

2.7.6 Design patterns

During the design several patterns have been used and what could be considered new patterns have been created. The factory sphere and communication system

could be considered as new patterns because they work on generally any thing. But before this can be concluded a comprehensive analysis has to be performed. Ref [8] describes how design patterns could be identified. Before a code structure can be considered a pattern, it has to general enough to be used by any object with only a small modification. It also has to be implementable in more than one language and a usability study has to be performed. None of these things have been analysed in this thesis and the reasons for this is time. An analysis of the factory sphere and communication manager would take approximately 2 to 3 month.

A faster way could be trying to implement these as templates instead (or generic in C#). When implementing the factory sphere it was discovered that many design patterns are easy to implement as templates. It could be interesting to see which of these method best could describe a design pattern.

Chapter 3

Case studys

Contents

In this chapter cases will be discussed for framework usability regarding TCM related devices. The cases described have been kept small in order to keep focus on specific problems and solutions. Creating one large case study could obscure some of the details involved. The case studies will describe different problems identified in the design of each software module. They will also describe some of the concepts and requirements.

The chapter contains information about:

- How high-speed communication between modules can be made.
- How restrictions can be implemented into configuration files.

- How the communication system could be extended into a distributed system.
- How interaction between user interfaces could be performed.
- What happens when a protocol for a communication channel is evolved.
- How measurements could be controlled in a user interface.
- How the framework could be auto configured by detecting connected hardware.

3.1 Case A - High speed communication

This case describes the different options for communication supported in the framework. This case have been selected because section 2.4.8 describes performance problems in the communication system. SpO_2 requires high-speed synchronised communication, describing each available method should help further development in selecting the correct methods.

3.1.1 Problem description

Section 2.4 describes how communication between modules can be performed using communication channels. A problem using this system for communication could be performance. In most TMC related measurements performance should not be a problem since data is send with an interval of 2s. However there is one module where performance could be a problem.

The TCM40 device has a hardware module used to measure SpO_2 and pulse. This module can also send out graph data. The graph data sent from the SpO_2 hardware module is used to show the rhythm of the heartbeat. To depict how this rhythm, there is sent data for the graph with an interval of a few milliseconds. The SpO_2 can send wave information with the intervals 79Hz, 38Hz, 25.3Hz or 19Hz depending on configuration. This gives the system min 13ms and max 52ms to handle the graph value. As default the hardware module sends data every 13ms this is also used by the current TCM40 devices.

It is also very importent that measurements are received in the correct order and with an even interval. This is because the curve has to be drawn at a specific speed of about 25mm/s. If graph data do not come with an even interval, it is impossible to draw the graph correctly.

3.1.2 Scenario

The framework gives 3 methods for communication. These are synchronous and asynchronous communication through the communication system. The last method bypasses the communication system and creates a bridge between the modules.

This case will use a SpO_2 module configured to send data every 52ms. This has been select because this is the system have to run on the emulator device.

Before a method can be used it must fulfil the following requirements:

- Graph data must come with an even interval of 52ms or less.
- Graph data come in the correct order.

3.1.3 Solutions

3.1.3.1 Method 1

As default the communication system used asynchronous communication between software modules. This method cannot be used because asynchronous communication cannot guarantee either of the requirements. Measurements can come in a different order than they are measured. The time interval for receiving the measurements can also very, depending on the load of the system.

3.1.3.2 Method 2

Communication channels can be setup to transmit communication synchronously. This guaranties the measurements comes in the order they are measured and also measurements come in an even interval. Using the communication system could however give problems on low-end systems. The problem is data have to be packed and unpacked from an XML document and this takes time. Low-end system could have trouble creating these XML documents within 52ms and still have an acceptable safety margin on resource usage.

Creating a system where some modules require 99% of the resources are not acceptable because other parts of the system then easily could disrupt the data flow, if they suddenly require a large amount of resources.

Using this method is acceptable but is requires an analysis of the performance on the desired platform.

3.1.3.3 Method 3

This method uses the factory sphere to create a bridge between the SpO_2 software measurement module and the software module used to display the SpO_2 values.

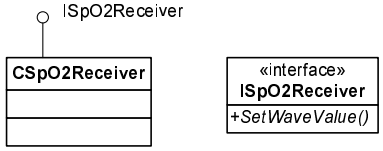

Figure 3.1: Bridge interface and object.

Figure 3.1 shows the interface and object defined to the bridge between the two modules. The interface is shared between the modules and the object is registered into the factory sphere. The interface have a function used to set SpO_2 value, the object is responsible for implementing and handling the values.

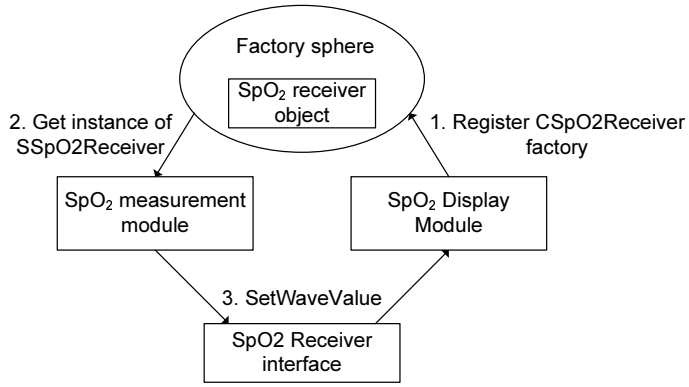

Figure 3.2: Connection modules through bridge.

Figure 3.2 shows an overview for how the bridge is created.

1. When the system is initialized the view responsible for displaying the wave values register the object "CSpO2Receiver" object into the factory sphere.
2. This allows the SpO_2 measurement module to get an instance of the object.

3. Using the shared interface on the instance received allows the measurement module to set SpO_2 values into the object. The object can then function as a bride between the measurement module and the display module.

This method is a little more complex to implement than using the communication system. However because the SpO_2 data do not have to be packed into an XML document, makes this method a lot more efficient.

3.1.4 Summery

Method 2 and 3 are both acceptable solutions for the problem and can both be used depending on performance requirements and capabilities of the system.

3.1.4.1 Method 2

This method should be used if possible because this method makes the system more flexible. The method allows transmission to more than one module without creating any bindings between these.

Qualities

- No extra implementation required.
- No extra bindings or dependencies on other modules.
- Send data to multiple sources.

Drawbacks

- Worse performance compared to method 3.

3.1.4.2 Method 3

This method creates a weak binding between the modules because they need the interface for the bridge object. There is also some dependency on the sequence of initialization. The measurement module cannot get the bridge object before the display module register it. The shown example can also only send data to one other module. The communication module allows for data to be transmitted to

multiple listeners. To achieve this using the factory sphere as a way to bridge the modules would require a more complex implementation of the "CSpO2Receiver" object. One way to solve the problem could be by letting the object implement an observer pattern [1]. This would require all the modules to share the interface creating a weak bound between them.

Qualities

- Better performance than method 2.

Drawbacks

- More complex implementation.
- Creates dependency and weak bindings between modules.

3.2 Case B - Configuration security

This case has been selected to show the different options available in the configuration system described in section 2.5. Showing the available options should help in selecting the correct type of configuration file in the future.

3.2.1 Problem

Section 2.5 describes how the configuration uses a digital signature to verify the integrity of configuration files. There are several types of data that could be interesting to store an external configuration files. These could be default data for modules, user configuration, and license information. License information's could be the number of allowed modules of a specific type and other information used to restrict module functionality.

This means some of the configuration files must not be changeable and others should be changeable.

3.2.2 Scenario

The case will describe 2 types of configuration files. The first type is called a dynamic configuration file. This type of configuration file is a file where the framework is allowed to alter data. The second type of configuration is a static configuration file where the framework cannot alter data.

3.2.3 Solutions

3.2.3.1 Dynamic configurations files

Dynamic configuration files are containing parameters the user is allowed to adjust. These parameters could be printer setup or alarm limits for O_2 or CO_2.

Dynamic configuration files require a private RSA key to be preset in framework in order to generate a new signature when the file is saved. For this reason the signature in this type of configuration file, is for integrity test only. The private key could be extracted from the system and used for manually updating.

This makes dynamic configuration file unsuitable to store static data, but it allows for storage of user changed data.

Qualities

- Detection of corrupt disk.
- Attributes can be changed in the configuration files.
- Attributes can be added.
- Attributes can be removed.

Drawbacks

- Limited detection of external editing.

3.2.3.2 Static configuration files

Static configuration is designed to hold license information used to limit functionality in the framework. But they can also be used to hold factory default

values and other parameters that are not allowed to be changed. License information's could be information's about how many O_2 hardware modules that is supported by the framework. It is not desirable to let a costumer use more O_2 sensors the than they have bought licenses for. Another example could be to limit the number of sensors allowed in the system because of user interface limitations.

The main difference between the static and dynamic configuration files are, the static does not need a private RSA key to be present. This means the framework cannot change the configuration files. This means not only do the system offers detection of corrupt disks it also offers detection of external editing. Since the private key is not necessary to have on the system, like it is for the dynamic configuration file, makes it difficult to edit the files because the signature cannot be updates.

3.2.3.3 Qualities

- Detection of corrupt disk.
- Detection of external editing.
- Cannot be changed.

3.2.4 Drawbacks

- Only the files owner can change it.

3.2.5 Summery

The configuration system gives a way to control license information in the application. This is done in a way that makes it difficult for the costumer to alter. The configuration system also gives a way to let the user store a system setup on the disk in a secure way. Meaning the configuration system detects if the setup have been damaged by a corrupt disk and for limited external editing.

The system does not give a way to store data encrypted on the disk. This means if there is sensitive data in a configuration file, then this data can be read by anyone.

3.3 Case C - Distributed system

This case have been selected to show the modularity of the framework, when software modules use the communication manager described in section 2.4 to interact. The framework was not indented to become a distributed system, but the designed system allows it. The purpose of this case is to show how this is done and some of the problems that have to be looked at in future development.

3.3.1 Problem

How the next TCM system will be is currently not known. This means the future system could be a distributed system where modules are located on more than one TCM device. This case explores how a distributed system could be created using the communication system described in section 2.4.

3.3.2 Scenario

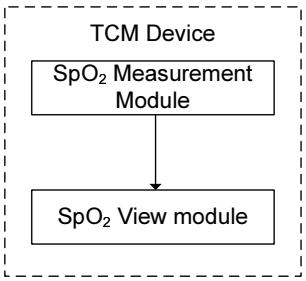

Figure 3.3: Standard module setup.

Figure 3.3 shows how a normal device could look. The system contains a module for measuring SpO_2 and a module that can show the measurements on the screen. These modules are connected using the frameworks communication system. The system will be used to demonstrate how modules could be distributed to more than one device.

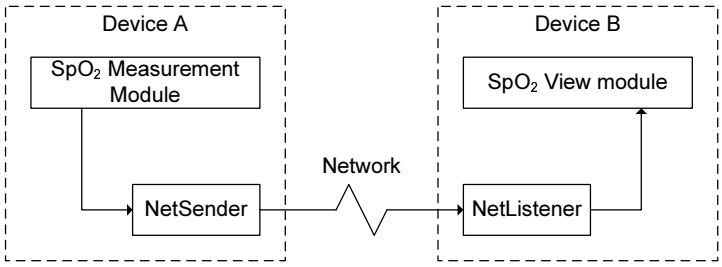

Figure 3.4: Distributed module setup.

3.3.3 Solution

Because the modules only know the protocol and channel used, makes it relative simple to split up the system into 2 different devices. Figure 3.4 shows how the system could be split. Device A in this system contains the SpO_2 measurement module. The communication channel where it sends measurements out on a network sender is now connected instead of the view module. Device B contains a network listener that receives packages sent from Device A. These packages are then send to the view module, displaying the SpO_2 measurements on a display.

The two SpO_2 modules do not care how data are transmitted between the modules. As long as they sends and listens to a SpO_2 channel they can communicate. What is in between is transparent for the modules.

This method could be used to create small and cheap measurements systems placed locally at the patient's side. And a central unit used to monitor all patients.

3.3.4 Qualities

- The communication system makes the transport media transparent for the modules.
- The same modules can be used in a compact and distributed system.

3.3.5 Drawbacks

- Net senders and listeners can be very complex to implement regarding security requirements.

- Not all measurement type can be sent over all type of transport media.

3.3.6 Summery

The current framework has a prototype network sender and listener that simply takes the protocol received and transmits it directly over the Internet. There are a lot of security considerations that have not been taken into consideration regarding sending data over a network. Also some of the measurements made by the modules cannot be sent over all types of networks.

The SpO_2 module sends a graph used to show the heart rhythm. Data for this graph must be displayed within a specific time interval in order to show the rhythm correctly. Sending data for this graph over the network could result in a graph that cannot be shown correctly. There is also an issue about the time it takes to transmit data over the network. The purpose is to send medical data used in treatment. Harm could come to a patient if data is received too late.

Some module types would however be beneficial to have located in a central place. These could be modules responsible for storing measurements, where the time is takes to transmit data is not important. Also modules responsible for controlling encryption keys could be beneficial to have located in one place.

3.4 Case D - Interaction between user interfaces

This case was selected to show how interaction between views in the user interfaces could be implemented into the framework. Section 2.6 describes how views are controlled and the limitations there are in the views. Interaction was not a part of the analysis done during the development of the system, but should have been a requirement. The purpose of the case is to show how an external system and the framework can be used to implement interaction.

3.4.1 Problem

Chapter 2.6 describes how the views in the user interface interacts. The interaction supported by the framework is very limited. This is however a problem considering views some time has to share datasets and modify these in. Also disabling buttons could be a problem when not knowing anything about other

views.

3.4.2 Scenario

A common problem in TCM could be the control of patient Id. In TCM4 system it is possible to enter a patient Id. This Id is shown on all the views displaying measurement data. In the TCM400 system it is also used to control measurement sessions.

3.4.3 Solutions

3.4.3.1 Using communication channels

One method for creating interaction between the views could be by using communication channels.

A view could create a channel and use this channel every time a change to a dataset has been performed. Any one interested in the data could then listen to this channel and get updates.

This is however not preferred since all views that have data to exchange need to create a channel and send data over. Changing the rules for notification of changes is difficult because changes have to be made in all the views. This would also give some undesired bindings between the modules since they have to communicate directly if values have to be modified.

The preferred way would be to implement an Observer pattern to handle this type of traffic.

3.4.3.2 Observer pattern

This section describes how an observer pattern can be implemented into the framework.

A Common way to create interaction between views is to create an observer that could hold the required data and then notify the views when data is changed. An example of an observer pattern can be seen in ref [1] page 293. The design to be used is very close the one described in ref [1].

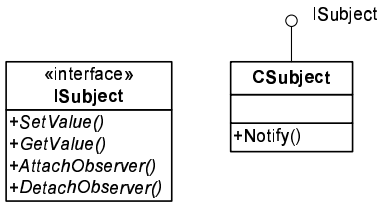

Figure 3.5: Interface and object for observer implementation.

Figure 3.5 show the object and interface needed. The implemented subject object should be shared through the factory sphere system. This would give all the views access to the object. Views can then through the interface access the subject object and register observers.

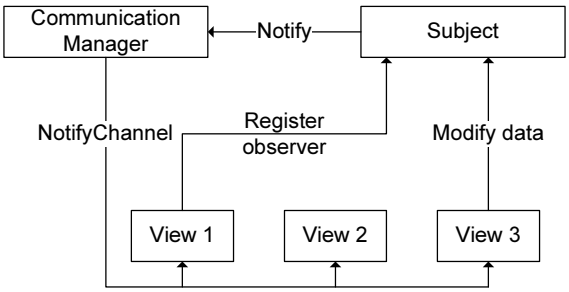

Figure 3.6: Structure of observer pattern implementation.

Figure 3.6 give an overview of how it should look. The system uses the communication system to send notification messages to the views. This method has been selected because the subject objects need a list of observers to contact when values is updated. The communication system has a system that can be used for this so instead of implementing a new system to send a signal to multiple listeners the communication system is used. By letting listeners to the channel be registered through the subject object makes the channel used transparent for the views.

3.4.4 Summery

Using the observer pattern the given structure on figure 3.6 allows for exchange of the subject object if different notification rules are needed. This can be done without having to recompile the views. The structure is also very similar to the original observer pattern described in ref [1].

The bindings between the modules have not been increased since they have to communicate with the subject object. The system is also better than using the communication system directly because the values are stored in a central place.

The subject object has to be implemented for each type of interaction required. This could mean that there should be a subject object for disabling buttons, and subject objects for different datasets.

3.4.4.1 Qualities

- No bindings between the views.
- Transparent usage of the communication system.
- Allowed other objects to interact with multiple views.

3.4.4.2 Drawbacks

- Different implementations of subject object might be needed depending on the systems configuration.

3.5 Case E - Evolution on communication channels

This case has been selected to show how the concept of evolution and slippage problems described in ref. [3] affect the communication system described in section 2.4. The purpose of this case is to show the effect module evolution have on the system. Since communication between modules should only be performed using the communication system means this is where evolution problem could show them self.

Evolution is defined as system object changing the way they communicate with other systems.

Communication is done using XML based protocols. The following example will be used to show how evolution can affect the communication in this system.

```
<Protocol>
```

```
  <SpO2Value></SpO2Value>
</Protocol>
```

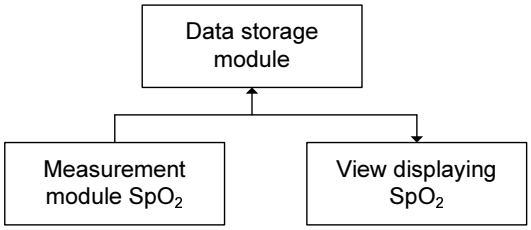

Figure 3.7: Modules linked through a communication channel.

The system used is shown on figure 3.7. The system shows a SpO_2 module sending out measurements to a storage system and to a view displaying the results.

Evolution can go two ways in the communication system. Attributes can be added to the protocol due to new parameters. Attributes can be removed do to obsolete parameters.

3.5.1 Adding new parameter

An attribute called BPM could be added to the protocol allowing it to transmit the SpO_2 pulse parameter.

```
<Protocol>
  <SpO2Value></SpO2Value>
  <BPMValue></BPMValue>
</Protocol>
```

Considering the storage and view module currently do not know the attribute "BMPValue" means this attribute is simply ignored. For the modules to use it they need to be upgraded. This means if an attribute is added to a protocol a combination of both old and new modules could be used at the same time.

As an example only the storage module is upgraded to support the new parameter. Both modules would still be able to handle the new protocol. Retesting is only required of the storage module.

3.5.2 Removing a parameter

Both storage and view module are now upgraded to use both attributes in the protocol. Now the SpO_2 parameter will be removed from the system.

```
<Protocol>
  <BPMValue></BPMValue>
</Protocol>
```

If the storage and view modules got this protocol they would both fail because they expect the SpO_2 module. When designing modules they should be prepared to handle situations where one or more parameters are missing. This is to ensure that one defect module cant crash the system because is sends defect packages out on a channel.

3.5.3 Summery

Attributes can easily be added to the protocols without having to upgrade all the modules using the protocol. This makes it easy to extend the system. Removing attributes however can become a very complex affair since all the modules have to be upgraded to use the new protocol.

Using normal interfaces between the modules instead of the communication channels would require recompilation of each module and retesting every time an attribute is added or removed. The same would apply if a static data structure were used.

3.5.3.1 Qualities

- Easy to extend protocols.
- Extension requires minimal amount of testing.

3.5.3.2 Drawbacks

- Difficult to reduce protocols.
- Reduction requires complete retesting of all affected modules.

3.6 Case F - Controlling measurement location

This have been selected to show how the configuration manager described in section 2.5 can be used to control where the measurements are shown in a view and what have to be implemented in the future. The purpose of this case is to show how to control measurements on the user interface.

The system can have more than one sensor of the same type located. The current TCM device can have up to 6 O_2 measurement modules connected at the same time. A method is needed to ensure the measurements from specific modules always are shown in the same position on the screen.

In the current TCM system measurements are shown in the screen based on the port it is connect to. The method is possible to use on the TCM device because each module have a dedicated serial port. However this system cannot be used if the TCM device uses another communication system. If the modules are connected through USB or Ethernet then it could be a problem to ensure the same module always has the same position on the screen. This is because the not necessarily can be identified on the port used, because these can be changed every time the system starts up.

The measurement protocol contains a "HardwareID" attribute than can be used to uniquely identify each hardware module. The identification can be done using the hardware modules serial number.

By having a unique identification of each module it is now possible to create a configuration file describing where each module should be shown on the view.

3.6.1 Qualities

- Modules can be uniquely identified.
- Can control modules placement on the view.

3.6.2 Drawbacks

- Currently it is not all modules that contains a serial number.
- Require reconfiguration when modules are changed.

3.6.3 Summery

The solution is relative simple using the module serial number. Currently there is one problem using this method. That is the SpO_2 module does not support retrieval of the modules serial number. However normally it is not necessary to use more than one module when measuring SpO_2 and pulse. O_2 is the most common modules used when it comes to using multiple sensors. The reason for this, is this type of sensors are used in wound treatment where several sensors are placed around the wound. The O_2 modules do have a serial number that can be retrieved.

The biggest problem would be if modules are changed for a new one. A change like that would require a reconfiguration of the view. For this reason it should be considered if a tool for reconfiguring should be implemented into the system.

3.7 Case G - Auto configuration

This case have been selected to show how a future system could be if concepts described in section 1.4.5.2 were to be implemented and the overall requirement OR12 were to be removed. This case will discuss how the system could be automatically configured if two of the concepts in the preliminary analysis was implemented. The concepts are "Hardware Abstraction Layer" and "Hardware Discovery".

3.7.1 Hardware Abstraction Layer

The preliminary analysis section 1.4.5.1 contains a concept called the HAL was shown. This is not something that has been discussed during the design, but is a relative important part for a larger system.

Figure 1.1 in the preliminary analysis shows the concept of the HAL layer. This is a small abstraction layer between the software modules and the hardware. The purpose of the layer is to have an abstraction between the software and hardware allowing the software to communicate with the hardware without knowing how the communication is performed. This means there could be several different hardware versions of O_2 modules each using different hardware interfaces. Since the software module do not communicate directly with the hardware makes it possible reuse of the same software modules. Changing hardware should result in a reconfiguration or a new HAL layer only. The main purpose of the HAL

layer is to contain a list of hardware ports and rules to communicate with these. It should also contain a number of bridges for the software modules to allow them to communicate with the hardware.

3.7.2 Hardware discovery

Figure 1.2 in the preliminary analysis have a concept called Hardware Discovery (HWD). The purpose of this module is to identify the hardware module connected to the device. This module is closely linked to the HAL layer. It contains rules for identifying different hardware modules.

3.7.3 Bridge objects

In order for the software modules to communicate with the hardware a number of bridge objects are needed. Figure 3.8 shows how these could be designed.

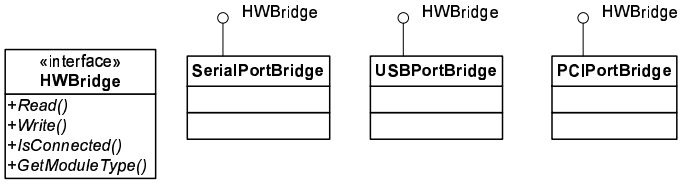

Figure 3.8: Bridge objects for HAL layer.

The bridge objects implements a common interface allowing a unified way of communication to a hardware device. The interfaces shown here, contains read and write functions used to send and receive data from a module. Each of the shown objects implements the "HWBridge" interface, this hides any specific hardware information's from the software modules. This allows the software modules to use any type of hardware communication.

3.7.4 Stating HWD

Once the HAL layer and the HWD module have been initialized they can start the discovery process. The process is as follows:

1. The HWD asks the HAL layer of a list of available hardware ports.

2. Create a bridge object for each port.

3. Try to discover the identity of modules connected to each port.

4. Bridge objects connected to a hardware module is registered in the HAL layer.

The end result for this procedure is a list of bridge objects in the HAL layer. These bridge objects allow the software modules to communicate with the hardware.

3.7.5 Using HAL and HWD to start the framework

Up till now it has been shortly described how hardware modules should be identified and how software modules should connect to these. The software modules implemented for a TCM device would in most cases be distributed as one large software package. This package not only contains all the modules implemented, but also a number of configuration files describing how the software modules can be combined. There could be a configuration file describing a TCM400 device or a TCM4 device. The result from the HWD could be used at startup to select one of these configurations. This would allow the application to automatically start the needed modules for a specific configuration.

Consider the HWD discovers 6 O_2 modules connected to the device. Using the current TCM devices as a reference this would then be identified as TCM400 device. The HWD could then lookup in a rule list to se if there is a valid configuration file for the found modules. A rule for a TCM400 device could be only O_2 module found and only between 1 and 6 modules allowed.

Given the HWD now have identified the device type it could give the TCM400 configuration file to the framework. The framework would then start loading the appropriate modules.

A rule for TCM4 device could be only one combo module found. If this was the case the HWD could give a TCM4 configuration file to the framework and it would start as a TCM4.

The current TCM devices rely on a technician to manually configure the device to a specific type, depending on the modules connected. This means if the modules were changed to reflect a different TCM device, the system would have to be manually reconfigured to reflect the changes.

3.7.6 Qualities

- Automatic recognition of device type.
- Automatic loading of required modules.
- Easy to add new device types to the system.
- No hardware bindings between the hardware module and software modules.

3.7.7 Drawbacks

- Complex to implement.
- Identification could be time consuming.

3.7.8 Summery

The system could be relative complex to implement depending on the number of different type of communication options that have to be available. Also a relative large number of configuration files have to be available depending on how many different combinations of hardware modules that are allowed. The discovery process could also be an issue if the HWD has to ask all available ports for all types of hardware modules. This could especially be a problem at startup time. Consider a system containing 10 ports. There are 10 different hardware modules. If it takes 6 seconds to search for one module it could in a worst case take up to $10 * 10 * 6/60 = 10$ minutes for the search to complete. In most situations this would be an unacceptable startup time. This could be solved implementing rules describing the type of hardware a specific hardware module can be connected to. Also implement an efficient identification system into each module could help. This is currently not available in the hardware modules, but should be considered for the next generation.

The biggest advantages using the system are the communication between the hardware and software modules have been separated. The bridge objects allowed different connection type completely invisible from the software's point of view. The HWD's value however could be discussed. Its biggest advantage is the system automatically can configure itself depending on the connected hardware modules. However the current devices in existence are selfdom reconfigured once sold to a costumer. This means the time used to configure the devices is

relative small. However if a future system is meant to become for fluent than the current system it could be considerer as a possible solution.

Chapter 4

Conclusion

Morten S. Sabinsky S973936

This chapter describes the overall results of the thesis. It contains

- Description of the overall results and what has been accompliced.
- A description of the results for each designed module.
- Description of items to be looked at in the future.

4.1 Overall results

This thesis have been used to design and implement a "State of the art" framework for an embedded system. The framework currently consists of software modules that handle object sharing, communication between software modules, configuration files and user interfaces.

The implemented modules shows that the theories and selected methods can be used. The theories and methods have to be refined before they will be

usable in a medical grade application. Some of the refinements that should be performed for the designed software modules are described in the next sections. During the refinement process an evaluation on each software module should be performed. The evaluation should result in an overview of cost/benefit regarding the software modules. If the software module should become over complex to implement and offer only limited benefit for the overall solution they should be removed. There are already concerns regarding the user interface system. This is a good example where the design may give more problems than is solves. In these situations a more traditional design could be used, where severe components are statically linked together. As an example all the views are linked together an functions like one software module instead of each view is a software module.

As shown in the final time schedules for the thesis in appendix H, the overall time consumption for thesis has been following the predicted schedule. The reason why it have been possible to follow these schedules so precisely is mainly because of the challenges solved in the start of the process, also having the concepts as guidelines during the module design have been helpful. However the concepts might also have limited the creativity process a little, because it was difficult to move focus away from these.

Overall the thesis shows new ideas and interesting solutions for both object sharing and communication between modules. The current state of the framework allows the framework modules to be exchanged without having any direct effect on any other modules. The way the system is designed allows for easy evolution of modules. The communication system allows modules requiring different protocol versions to function together as long as backward compatibility is maintained. The factory system allowed modules an easy way of sharing objects without the framework have to be recompiled to support these. The designed configuration system allows many of the dependencies in the system to be located in configuration files and not in source code. This means most of the code dependencies have been removed.

4.2 Design results

4.2.1 Factory Sphere

The factory sphere descibed in section 2.3, is a software module that control how objects can be shared between modules. The system works by allowing modules to register a factory object. Other modules can then use the factory objects to create an instance of a specific object.

In its current state, it is fully functional. There are some issues that needs to be looked at:

Performance
Performance is an item that should be looked at once a hardware platform has been specified.

Factory removal
There are also some issues regarding removal of factories from the sphere. Currently this is not allowed because there are no methods to handle this safely. However if this could be handled safely a hole new area of usability for the factory system could be reveled.

4.2.2 Communication Manager

The communication manager described in section 2.4, is a software module that is used to control how modules communicate with each other. The Communication manager creates channels between the modules where protocols can be transmitted.

In its current form, it is fully functional, but there are some issues that must me looked at:

Performance
Currently a reference to a parsed XML document is sent to each listener. This means data have to be parsed in to text format and from text format to be used. An exploration for a better method for data transmission should be explored.

Data integrity
Listeners can currently modify data in the received XML documents. Since the listeners only receive a reference, a modification could influence other listeners. This is a problem because this could lead to unpredictable results.

Distributed system
As described in section 3.3 the communication system could be used to distribute modules over several devices. It could be extremely interesting to map and explore problems and usable cases for this. It is not some that have had the direct focus in this thesis, but it is possible to implement with only minor modifications.

4.2.3 Configuration manager

The confuguration manager descibed in section 2.5, is a software module that contains a simple implementation allowing load and save of XML files and signature verification and generation. Its current form if fully functional, but there is one item that should be explored. This could be some type of basic encryption of the data stored in the configuration files.

4.2.4 Controlling user interfaces

This system is descibed in section 2.6 and is a system that controls the user interface is a reduced version of the application controller pattern described in [2].

The implemented module works, but there are some issues regarding dependencies between the modules that have to be solved.

This implementation requires a lot of documentation. Not only must every implemented user interface be described. But also the state machine controlling the user interfaces must describe all interaction.

This might not be the best solution. A better solution could be to create a complete software package where each view is statically linked. In most cases the user interfaces would always be configured in one specific way. Once configured it will seldom change. The reusability of that is gained using the application controller pattern, might not be more cost effective than implementing a normal user interface, because of the documentation amount.

User interfaces that have been carefully designed can have views statically linked to each other, without this gives any problems.

4.3 Case study conclusion

The cases study has been used to show how the software modules can be used and what should be considered for future development. The purpose of this section is to shortly describe and summaries the results for each of cases shown in the case study.

4.3.1 Case A - High speed communication

Case A described in section 3.1, is used to described how the factory sphere in section 2.3 and the communication manager in section 2.4, both can be used to establish communication between software modules.

The use case describes 3 methods available in the framework and discusses the usability regarding transmission of data from a SpO_2 software module. Method 1 cannot be used since this is an asynchronous method and do not guaranty measurements come in the correct order and at an even interval. Method 2 can be used, but this method could have some performance issues that have to be solved. By method 2 is the preferred method because it uses the communication system. Method 3 uses the factory sphere to create a bridge between the software modules. This method offers the best performance, but creates a dependency between the modules that have to communicate. This dependency can give rise to slippage problems at a later time.

4.3.2 Case B - Configuration security

Case B described in section 3.2, is used to describe how configuration files can be secured using a digital signature. The case describes the different types of configuration files (static and dynamic) and what purposes the configuration files are bested suited for.

Static configuration files should be used to store default configuration data for modules. They are also suitable to store license restrictions in because they cannot be altered. Dynamic configuration files are however only suitable to store user controlled data. They are less secure that static files because the private RSA key used to generate the configuration files signature, have to be stored on the TCM device.

4.3.3 Case C - Distributed system

Case C described in section 3.3, is used to describe how the framework can be distributed if the software modules rely on the communication manager in section 2.4 for communication. The case described what is necessary to implement in the current framework and what considerations that have to be done for a future system.

The framework can easily be distributed if modules only rely on the communication manager. Section 4.4.4 describes some of the problems that have to be considerer before this is done.

4.3.4 Case D - Interaction between user interfaces

Case D described in section 3.4, used to describe how interaction between views in the user interface is possible. As described in section 2.6 this is not possible in the current design of the views. But a combination of an external module sharing a subject object using the factory sphere and using the communication system to notify views, allows complex interactions. The system shown in the case study allows views to interact without communication directly with each other. This means there is almost no bindings between the views and they should be easy of exchange or exclude. A multiple number of these subject objects could be implemented allowing very complex interaction to occur.

4.3.5 Case E - Evolution on communication channels

Case E described in section 3.5, shows the effect evolution have on module using the communication manager to interact. This shows why it could be beneficial to use such a system, because adding parameters to the protocol do no effect modules using an older version of the used protocol.

4.3.6 Case F - Controlling measurement location

Case F described in section 3.6, show how the communication could be used to control the locations of measurements in a view. This could be a problem in a dynamic where it not necessarily knows where measurements are coming from. The case also describes how a tool on the device to configure the measurement locations in a view, could be helpful to have.

4.3.7 Case G - Auto configuration

Case G described in section 3.7, discuses how the framework could be auto configure it to load a specific set of software module, based on the connected

hardware module. The case is based on the concepts described in section 1.4.5.2, specifically the concepts about Hardware Abstraction and Hardware Detection.

The case main purpose was to map out how the framework will function after these modules are implemented.

4.4 Future development

As described in the earlier in chapter 1.3.4 the model implemented in this thesis, is only an evaluation model. This also means the list of future improvements are relative long. This section however will only show some of the most important improvements needed if this framework is to be used.

4.4.1 Performance

Several times during the development of the core modules performance have been mentioned. The framework design has not taken into consideration the type of hardware platform it should run on. This is because the profile for a new TCM device is not developed. Once this is known performance must be tested.

Some of the initiative that could be taken to increase performance could be by selecting different methods for transmitting data. Also using different methods for storing and identifying channel and object factories.

4.4.2 User interface

The system that controls the user interfaces should be looked at again. The reason for this is to streamline the dependencies between the user interface module and the control system. The way the dependencies are between the user interface modules and the control system are currently not the most optimal.

4.4.3 Is .NET the best framework?

The framework has been developed to run using .NET Compact Framework 2.0. The framework gives access to a large number of tools that makes software

development easy. However there are a few problems when using the .NET framework. The first problem is access to external hardware. The support for this in .NET is currently limited to serial ports. This means any other type of access could be difficult to implement into the application. The framework itself does also take up space on the disk. Depending on the platform used for the next generation TCM device this could be a problem. Moving the final application from one OS to another can also be a problem. The problem is mostly when trying to move user interface components. If an OS like Linux or BSD is used, then the mono framework could be used. However this framework do not support the user interface components. An alternative could be to use GTK#[1] for user interface modules, but license requirements should be explored before any usage.

4.4.4 Distributed system

As described in the case study in section 3.3, modules that used the communication system to interact with other modules can be distributed to other devices. However when the communication system was designed this usage was not a part of the considerations done. There are a lot of security considerations that have to be considerer before the framework is used in this matter. The following list contains descriptions of some of the considerations that are needed.

4.4.4.1 Online/Offline

Then devices are network connected there are a lot of external elements that can influence the channels used. This means that devices are not always capable of communication with each other. How modules or the framework should react when communication to an external device must be considered before usage. Should data be stored in buffers and if so how much data should be stored. How should module react when they are starved of data. Can the module function if not connected.

4.4.4.2 Security

Transmission security must be a required exploration before this is used. How to ensure data received is valid, encryption of transmitted data. Recognition of devices is just some of the things to be explored. Allowing devices to be

[1] A free open source user interface system.

connected to a network also brings up the question of how to make it resistance to network worms and other types of attack. This is something that highly relies on the OS chosen.

4.4.4.3 High-speed synchronous communication

Hardware modules like the SpO_2 module sends graph data continuously. For those graphs to be shown correctly requires synchronous communication and enough bandwidth for the data to be transferred. Distributing modules like this over a network is almost impossible since there is no control of it. This means restrictions on how the system can be distributed must be specified.

4.4.5 Communication system

The designed communication system allows all modules to listen to any channel on the system. It should be considered if some type of restrictions should be implemented on this. The reason for this is allowing confidential data to be transmitted between modules. This protects sensitive personal data, propertary protocols or restricts access to certain functionality.

4.4.6 Factory system

The factory sphere is used as a global factory system for modules in the framework. Like the communication system all modules have access to the factory system. Here could access restrictions to certain objects be used to protect the system from modules accessing certain functionality. Like the communication system this protects sensitive data and restrict access to some functionality.

4.4.7 Using C#

The framework designed has been implemented using C#. Most of the current software written to the TCM devices is written in C/C++. This makes it difficult to directly reuse code. This forces the developers to rethink and rewrite the code so if fits better into the framework, but it is also much more costly to do this. Using C# should also depend on the usage of the .NET Compact

Framework. If this were not to be used, it would be better to choose an implementation done in C/C++. Linking unmanaged code to the framework could solve some of these problems.

4.4.8 Hardware abstraction layer

The hardware abstraction layer (HAL) is one on the concepts in the preliminary analysis that did not make it into this thesis. Currently the modules have to communicate directly to an external port. The purpose of the HAL layer is to separate the software modules from the hardware. This is a part of the framework that must be designed and implemented if the framework is to be used. Shielding the software modules from the hardware would make it easier to reuse the software module in a future system.

Appendix A

CD content and Glossary

A.1 CD content

This appendix describes the content of the supplied CD.

/Viola-print.pdf
This is an eletronic version of the printed thesis.

/Viola-net.pdf
This is an electronic version of the thesis meant for network distribution.

/Source/Prototypes
This directory contains prototype solutions created during the challange analysis and the design.

/Source/Viola
This directory contains unit test modules created for implemented modules.

/Source/ViolaV2
This directory containes the implemented software modules.

/Papers
This directory containes electronic versions of some of the papers used.

/PlatformFiles
This directory contains some of the custom files required by the created CE image.

/Misc
This directory contains drawing, documents and the latex version to this thesis.

A.2 Glossery

GUID	Global Unique Identifier. A 128 bit value
BPM	Heartbeats per minutes
CO_2	Carbon dioxide blood gas parameter
FDA	Federal Drug Administration
HAL	Hardware Abstraction Layer
Hardware Module	Hardware used to perform some type of measurement
O_2	Oxygen blood gas parameter
Object	Class or software module
Pulse	Heartbeats per minutes or Beats per munites
RqId	Requirement ID
Software Module	A number of objects describing a specific functionality
SpO_2	Oxygen saturation parameter measured in %
TCM	Transcutaneous monitoring
US	United States Of America
Neonatal	Child born to early
Runtime	The application is currently executed
Non-specific object	Object have no type
HWD	Hardware Discovery

Appendix B

Platform and Visual Studio 2005

The purpose of this chapter is to describe how connectivity is established between Visual Studio 2005 and a CE device over an Ethernet channel. The description below is based on information found in the help files of Visual Studio 2005 and the Platform Builder and the compressed into this small document.

As standard Visual Studio 2005 use Active Sync to communicate with CE devices. Active Sync is designed for CE client device, communicating over USB or a serial connection. The device this project uses does not have USB client compatibility and serial communication is not practical to use. The device does however have an Ethernet port suitable for downloading and debugging application, but using this requires a workaround of the Active Sync connection.

B.1 Platform requirements

When designing the platform, one library component must be included into the project. This is component is "CAB File Installer/Uninstaller". This component allows the installation of a CAB file generated by Visual Studio to be installed on the CE device.

Additionally the following files have to be added manually:

- ”clientshutdown.exe”
- ”CMAccept.exe”
- ”ConmanClient2.exe”
- ”eDbgTL.dll”
- ”TcpConnectionA.dll”

These files can on a standard installation of the Visual Studio 2005 be found at:
”C:\Program Files\Common Files\Microsoft Shared\CoreCon\1.0\Target\wce400\x86”

The reason these files have to be added manually is because we do not use active sync. Normally Visual Studio 2005 will download these files and activate these files if missing from the device. But since we du not use Active Sync, Visual Studio 2005 can’t see the CE device and download them.

Once the CE device is active we have to get the device current IP address. This can be done by typing ”s ipconfig /d” in the Platform Builders command console. ”s” specifies you want to start the application ”ipconfig” and the ”/d” redirects the application output to the Platform Builders command console.

B.2 Visual Studio

To establish communication with the CE device, we need to know the device IP address. Once the address is known we can in Visual Studio configure the target device.

1. Enter ”Tools” → ”Options”.
2. Open ”Device Tools” and select ”Devices”.
3. Select ”Windows CE 5.0 Device” and press ”Properties” button.
4. Select ”Use specific IP address” and enter the CE device current IP address.
5. Start ”\Windows\ConmanClient2.exe” on the platform.
6. Start ”\Windows\CMAccept.exe”on the platform.

Know you have 3 minutes to deploy an application to the CE device. Once an application is deployed within 3 minutes, you can continue to deploy application indefinitely.

Appendix C

Module GUID ID's

This chapter have a complete list of guid values used to identify modules.

Module class	Guid ID
CKeyManager	{49BF16AD-8CC5-43b2-9E00-8F2F199AF27B}
CModuleManager	{ED4C44CA-0DD0-4854-83D4-B5C6CA26E925}
CViewManager	{E05E0A5E-6B0C-49b8-800F-C65C1E448E62}
CConfigManager	{9DC957E3-E027-44d0-9005-2EC00F12A0C7}
CCommunicationCore	{E691CA8A-5C4E-4b8b-BAA8-B80E617CF133}

Table C.1: Module GUID ID's

Appendix D

Thesis description

This chapter containes the danish description of the thesis, used to getit approved.

D.1 Projekt oplæg til modulær TCM applikation V 0.4

Projekt location

Radiometer Danmark
Åkandevej 21
DK-2700 Brønshøj
Afd. 422 TCM Udvikling

Radiometer vejleder

Jørgen Belfalas
Tlf. 38273340
Email: jeb@radiometer.dk

Vejleder på DTU

Bjarne Poulsen

Tlf. 45255274
Email bjp@imm.dtu.dk

Projekt info
Projekt start 10/1-06
Projekt slut 31/07-06
Point 40

D.1.1 Problem stilling

På nuværende tidspunkt, har man 2 forskellige TCM applikationer, som man benytter alt efter hvilken hardware konfiguration man benytter. Ud over at applikationerne afhænger af hvilken hardware man benytter, så er der også relativ stor forskel i opbygningen af applikationerne, hvilket gør at man ikke direkte kan flytte funktionalitet mellem applikationerne hvis man ønsker dette. Dette kunne f.eks. være at man i TCM400 ønsker et interface til Vuelink som findes i TCM4/40. Eller omvendt man ønsker at benytte TCM400 mere avanceret datamaneger i TCM4/40.

Et andet problem med at have 2 applikationer, er den løbende vedligeholdelse. Da disse applikationer ikke er bygget ens op, betyder det, at man skal omstille sig fra et design til et andet. Dette er noget der tager tid fra udvikling processen, som kunne bruges bedre.

D.1.2 Løsnings forslag og mål

Dette ønskes løst ved at designe et state of the art framework som kan benytte komponenter fra et komponent bibliotek. Disse komponeter kunne være følgende

- Måleenhed som fortager måling
- En datamaneger komponent, som fortager data opsamling
- En kommunikations komponent som fortager kommunikation med eksterne enheder
- En IU komponent

For at kunne lave et sådan bibliotek, så må det kræves, at disse komponenter skal have en meget skarp grænseflade. En sådan grænseflade, vil også give den fordel, at man kan fortage unit test på hver komponent.

Målet med at have dette et bibliotek med komponenter, er at man via det designet framework kan modulært stykke en applikation man ønsker at benytte, sammen. Dette kunne være en fuld applikation, som kan alt, en håndholdt applikation, som kun benytter en elektrode komponent og et simpelt IU komponent. Eller en målestation som ikke indeholder en IU, men man kan kommunikere via en webserver i stedet.

Målet med frameworket, er at kunne loade moduler runtime, når der er brug for dem

D.1.3 Problem stillinger

Det er dog ikke uden problemer, at lave et bibliotek med komponenter. Følgende problemer skal der tages stilling til.

- Hvordan skal data flyde rundt i et komponent design
- Hvordan skal signalering af events fungerer
- Hvordan skal versions styring af moduler styres
- Hvordan skal moduler opdateres
- Hvordan sikres modulers integritet
- Skal tredje parts leverandører kunne levere komponenter (Hvis man må, hvad skal der så være af krav til et sådant modul)
- Hvordan sikres korrekt modul type (f.eks. hvis man har flere forskellige IU komponenter, hvordan ved man så, at det er den korrekte type)
- Hvad skal der være af krav til komponent konfigurering
- Hvordan en test strategi skal udformes
- Hvordan håndteres sprogvarianter

D.1.4 Ønsker og krav

Projektet ønskes udført som en .NET v2.0 løsning, hvor der benyttes C# og .NET compact framework. Som krav stilles det, at design af komponenter i det omfang det er relevant, benytter design patterns.

Appendix E

Protocols

E.1 Base document

This document is the foundation of all the documents and contains information about the system and protocol type.

```
<Base>
  <DeviceType></DeviceType>
  <DeviceSerial></DeviceSerial>
  <ProtocolID></ ProtocolID>
  <DeviceIP></DeviceIP>
  <PackageLifeCount><PackageLifeCount>
  <ReceiveIP></ReceiveIP>
  <ReceiveChannel></ReceiveChannel>
  ...
    Insert sub document here
  ...
</Base>
```

DeviceType
Describes the configured device identification. On the current system this

means if the device is a TCM4, TCM40 or TCM400. It could also be the name of the configuration.

DeviceSerial
This is the serial number of the device. Each device has a unique serial number used to identify the device. This ID can be used if data have to be collected from more than one source. This enables back tracing of data.

ProtocolID
This attribute is used to identify the contents of the protocol. The table below shows identified protocol ID's

Protocol Id	Description
Measurement	This is the protocol used to transfer information about a measurement
KeyFetch	This is used to request at key from the key manager system
KeyData	Used to return key data
DataFetch	This is used to request data from a storage system
Data	Used to return data from storage system

DeviceIP
This is the IP address of the device. This is only used in distributed systems.

PackageLifeCount
This is a number that specifies the number of times a package may parse a communication manager. The package is not retransmitted if PackageLifeCount <= 0. This attribute is to prevent data packages from being transmitted for an eternity if there is a loop in the communication system. Data packages dropped because of a low life count must be stored in frameworks log system

ReceiveIP
Optional attribute specifying a remote device to receive the package

ReceiveChannel
Optional attribute specifying receiving channel on remote system.

E.2 Measurement

This is a sub document inserted into the base document.

```
<Measurement>
```

```
  <ParameterName></ParameterName>
  <ParameterUnit><ParameterUnit>
  <Value></Value>
  <HardwareID></HardwareID>
  <LowerAlarmLimit></LowerAlarmLimit>
  <UpperAlarmLimit></UpperAlarmLimit>
  <Alarm></Alarm>
  <Timestamp></Timestamp>
  <SampleID></SampleID>
</Measurement>
```

ParameterName
: This is used to name the measurements type. Current valid names are O_2, CO_2, SpO_2 and Pulse.

ParameterUnit
: This describes the unit measurements are represented in. Current valid units are mmHg for O_2 and CO_2. SpO_2 are measured in % and pulse is in beats pre minutes.

Value
: This is the value of the measurements

HardwareID
: This is an identification field used to uniquely identify the hardware module. This could be the serial number of the measurements device. Currently it is not possible to get the serial number of all measurement devices. For that reason this ID is currently optional. The reason for putting this attribute into the protocol is the desire to be able to track a measurement to its source.

LowerAlarmLimit
: This is a value describing the lowest value allowed before an alarm is activated. Alarm is activated the value $<$ LowerAlarmLimit

UpperAlarmLimit
: This is a value describibg the higest allowed value before an alarm is activated. Alarm is activated when the value $>$ UpperAlarmLimit.

Alarm
: This is an optional field used to signal if an alarm is active. Valid values for this attribute are None, Low and High. This can also be derived using the given alarm limit.

Timestamp
: This field must contain the time when the measurements were taken. Format for the timestamp is YYYY-MM-DD hh:mm:ss.

SampleID
This attribute is to accommodate .NET compact frameworks "Date" object. Depending on the underling system the .NET framework does not guaranty a time resolution of more than one second. If more than one measurement is to be made during one second then they will get the same timestamp at we can use the sample ID to separate the measurements. The sample ID must be unique within one second. Every new sample ID must be incremented by the value 1.

E.3 Key fetch

This protocol is used to request encryption key data from a key manager.

```
<KeyFetch>
  <KeyID></KeyID>
  <ReturnPath></ReturnPath>
</KeyFetch>
```

KeyID
This is the ID of the key requested

ReturnPath
This is the GUID value of the module performing the request or channel name

E.4 KeyData

This protocol is used to distribute key data in the system

```
<KeyData>
  <KeyType></KeyType>
  <PrivateKey></PrivateKey>
  <KeyDataFormat></KeyDataFormat>
  <Key></Key>
  <KeyOwner></KeyOwner>
</KeyData>
```

KeyType
This is used to specify what type of key is used. Currently only RSA is supported

PrivateKey
Specify if the key contains private key data. Current values are "False" for public RSA keys and "True" for private RSA keys

KeyDataFormat
This described the keys data format. Currently only SNK format is supported.

Key
This attribute contains the key data. Key data comes in the comma separated hex numbers like this "AA,BB,10,11,12,FF"

KeyOwner
This is an optional attribute containing information about the key owner.

E.5 Data storage fetch

This protocol is used to request all measurement data for a specific time period using a specified time interval.

```
<DataFetch>
  <Starttime></Starttime>
  <Stoptime></Stoptime>
  <TimeInteval></TimeInteval>
  <SessionID></SessionID>
  <ReturnPath></ReturnPath>
</DataFetch>
```

Starttime
This attribute defines the start time for when data is to be retrieved. The time format is "YYYY-MM-DD hh:mm:ss". Data returned is where Starttime = Timestamp

Stoptime
This attribute defines the stop time for when data is to be received. The time format is "YYYY-MM-DD hh:mm:ss". Data returned is where Stoptime = Timestamp

TimeInteval
This attribute is used to describe the number of seconds returned measurement values should be interleaved with.

SessionID
This is an optional attribute than can be used to specify a specific session in the storage system. This should only be used if the storage system stores data in sessions.

ReturnPath This is the GUID value of the module performing the request or a channel name

E.6 Data

This protocol is used to send series of measurement data

```
<Data>
  <SampleCount>< /SampleCount>
  <Measurement ID=’’’’>
    ... Measurment block ...
  <Measurement>
</Data>
```

SampleCount
This attribute is used to describe how many samples are returned.

Measurement
This is a measurement sub document used to hold measurement data. The block has been given an ID to uniquely identify each sample. Contents of the measurement block can be seen under E.2 page 124. Data returned are sorted descending using ”Timestamp” as primary sort and ”SampleID” as secondary sort

E.7 Return path

When communicating between modules it is sometime necessary to give a return path where a response can be directed. For this reason one attribute is added to the base document and one the request documents.

In the base document there is an attribute called "DeviceIP". This attribute contains the devices IP address and is used if a request is directed to another device.

In the request documents there is an attribute called "ReturnPath". This attribute contains a channel where the requests should be directed.

The combination of IP and channel name makes is possible to send data to another device. The base document contains two optional attributes used to specify a remote receiver these are "ReceiveIP" and "ReceiveChannel". Using the device IP and return path from the request document we can initialize this two attributes a send the requested data back.

Appendix F

Thread memory leak test

The communication system creates a new thread every time an asynchronous call is made. Once a thread is created and started all references to it are thrown away by the communication system and the thread is left to itself. This method raised some questions about the garbage collectors ability to cleanup dead threads. For this reason I have made a small test that creates and starts 1000 thread at throws them away once started. I then monitor the memory usage to see if it remains stable over several runs. Figure F.1 shows a log created of the memory usage during the test. The yellow lines split the log into 4 sections. Each yellow line represents a new run where I creates and starts 1000 threads. The first section leading up to the first yellow line represents the first execution of the run. After this run I get a baseline of a 35% memory usage of the system. Each subsequent run of the test returns to the same baseline at 35%. From this I can conclude the garbage collector can cleanup thread correctly.

The test follows the procedure below.

1. Start garbage collection to remove all unwanted object from memory
2. Print out memory usage
3. Create and start 1000 threads and throw away the handles
4. Sleep to allow the threads to be executed

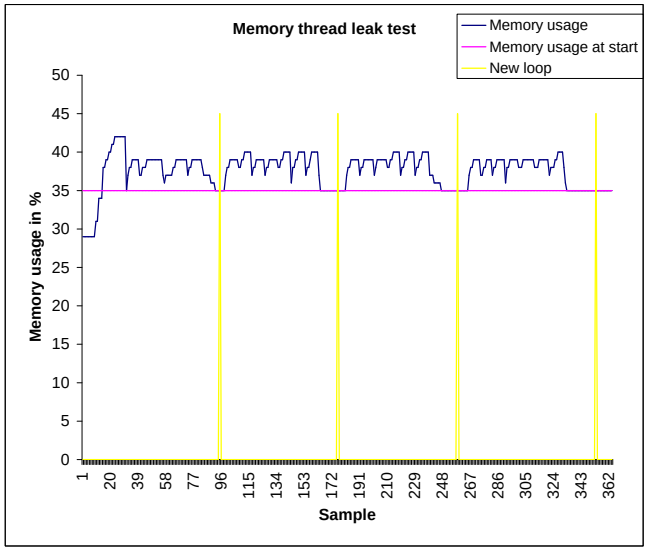

Figure F.1: Memory usage during leak test

5. Start garbage collection
6. Print out memory usage
7. Wait for restart or quit command

The functions called in the threads are just empty functions. This is to avoid too much cluttering of the memory that might obscure the results. During the test the memory usage was sampled every second and assigned a sample number. All measurement are supplied the CD.

Appendix G

Test protocols

G.1 Template for test protocol

Test name
Test case 1 to x

Test type
Level 1, level 2, Non-unit test or integration test.

Name of interface or object

Test steps

Step	RqId	Tested functionality	Expected result	Success/Failed
1				
...				
X				

Results of failed tests

Step	Actual result
X	

G.2 Test protocol for Communication Manager

Test name
Test of "ICommunicationManager" interface

Test type
Level 1

Name of interface or object
ICommunicationManager

Test steps

Step	RqId	Tested functionality	Expected result	Success/Failed
1	COM1	Create "ChannelA"	-	
2	COM4	Verify "ChannelA" exists	Channel exist	
3	COM1	Create "ChannelB"	-	
4	COM4	Verify "ChannelB" exists	Channel exits	
5	COM2	Add "Listener1" to "ChannelA"	-	
6	COM2	Add "Listener2" to "ChannelB"	-	
7	COM5	Send data to "ChannelA"	-	
8		Verify "Listener1" have received data	Data was received	
9		Verify "Listener2" have not received data	No was data reveived	
10	COM5	Send data to "ChannelB"	-	
11		Verify "Listener1" have not received data	No data received	
12		Verify "Listener2" have received data	Data Reveived	
13	COM2	Add "Listener3" to "ChannelA"	-	
14	COM5	Send data to "ChannelA"	-	
15		Verify "Listener1" have received data	Data was received	

Continued on next page

Step	RqId	Tested functionality	Expected result	Success/Failed
16		Verify "Listener2" have not received data	No was data received	
17		Verify "Listener3" have received data	Data was Received	
18	COM3	Remove "Listener2" from "ChannelA"	Error received	
19	COM3	Remove "Listener1" from "ChannelA"	-	
20	COM5	Send data to "ChannelA"	-	
21		Verify "Listener1" have not received data	No was data received	
22		Verify "Listener2" have not received data	No was data received	
23		Verify "Listener3" have received data	Data was received	
24	COM8	Add "SleepListener" to "ChannelA" as asynchronous	-	
25	COM7	Add "SleepListener" to "ChannelB" as synchronous	-	
26	COM5	Send data to "ChannelA"	-	
27		Verify "SleepListener" is executed after send is completed	Listener is executed after send is completed	
28	COM5	Send data to "ChannelB"	-	
29		Verify "SleepListener" is executed before send is completed	Listener is executed before send is completed	

Results of failed tests
No errors

G.3 Test protocol for Factory sphere

Test name
Test of "IFactorySphere" interface

Test type
Level 1

Name of interface or object
IFactorySphere

Test steps

Step	RqId	Tested functionality	Expected result	Success/Failed
1		Create instance of "CFactorySphere" object	-	
2	FAC8	Verify instance have "IFactorySphere" implemented	"IFactorySphere" is implemented	
3	FAC1	Register "TestFactory" using "TestFactory" as Id	-	
4	FAC2	Get instance using "TestFactory" as Id	Instance returned	
5		Verify returned instance is "TestObject"	Instance is of correct type	
6	FAC1	Register "TestObject" using the generic factory object "GTest" as Id	-	
7	FAC2	Get instance using "GTest" as Id	Instance returned	
8		Verify returned instance is "TestObject"	Instance is of correct type	
9		Save instance reference as "Test1"	-	
10	FAC2	Get instance using "GTest" as Id	Instance returned	
11		Verify returned instance is "TestObject"	Instance is of correct type	
12	FAC10	Verify returned instance reference is different from "Test1" reference	The instances are different	
13	FAC1	Register "TestObject" using the generic singleton object "STest" as Id	-	
14	FAC2	Get instance using "STest" as Id	Instance returned	
15		Verify returned instance is "TestObject"	Instance is of correct type	
16		Save instance reference as "Test1"	-	

Continued on next page

Step	RqId	Tested functionality	Expected result	Success/Failed
17	FAC2	Get instance using "STest" as Id	Instance returned	
18		Verify returned instance is "TestObject"	Instance is of correct type	
19	FAC11	Verify returned instance reference is the same as "Test1" reference	The instances have the same reference	
20		Init generic session object with the limit of 3 objects	-	
21	FAC1	Register "TestObject" using the generic session object "SesTest" as Id	-	
22	FAC2	Get 3 instances using "SesTest" as Id	3 instances returned	
23		Verify each returned instance is "TestObject"	Instances is of correct type	
24	FAC12	Try to get new instance using "SesTest" as Id	Error thrown	
25	FAC3	Release instance using the first returned reference	-	
26	FAC2	Get new instance using "SesTest" as Id	Instance returned	
27		Verify returned instance is "TestObject"	Instance is of correct type	

Results of failed tests

No errors

Appendix H

Time schedules

H.1 Thesis time schedule for introduction

	Weeks												
Project titel	2	3	4	5	6	7	8	9	10	11	12 - 20	21 - 29	30
Introduction/Startup	■												
Preliminary analysis		■	■	■	■	■							
Challenges							■	■	■	■			
Framework design											■		
Case study												■	
Correcting thesis documentation													■

Introduction/Startup
This phase is used to get an overview of the thesis and to create a schedule for the first part of thesis. Description of the thesis is generated.

Preliminary analysis
The preliminary analysis is used to get an overview of the methods to use, find relevant documentation, tools and brainstorm for ideas to finally generated concepts.

Risk Analysis
This part is used to analyze and find solutions to risks identified during the preliminary analysis

Framework design
8 weeks are set aside to for creating a framework design. A schedule for the framework design will first be created during the startup process of the design.

Case study
8 weeks have been set aside to make case studies of the framework. A schedule for this will be made during the startup process of the case studies.

Correcting thesis documentation
The last week of the thesis will be used to create and correct the final documentation for this thesis. Documentation for the individual parts of the thesis should be made during each process.

H.1.1 Actual time used for introduction

Introduction/Startup started week 2 9/1 - ended week 2 14/1
Completed introduction chapters and stated working on concept ideas

Preliminary analysis started week 3 16/1 - ended week 9 3/3
Concept generation
This was already stated during the approval period of the thesis. In the analysis phase the main purpose of the concept generation is to select the most useful ideas.

Technology/Tools/Methods
Most of the methods and tools were given in the thesis descriptions. Most of the work done in these sections was to find alternative tools to use. The sections was completed 30/1

Analysis of current theories
Most of the work done in the preliminary analysis is in this section. It was more time consuming to find relevant papers and books than expected. The reason for this is the limited number of relevant papers for this area of software development. This section was concluded 3/3.

Challenges started week 10 6/3 - ended week 12 23/3
This section describes the challenges that were solved before the main framework design started.

Test strategy
Base on the papers describing unit testing a test strategy was formulated. This was completed 24/3

Overall requirements and summery
These sections was completed 25/3

H.1.2 Summery

The introduction chapter took 6 days more than expected. The main reasons was the search for relevant papers and books took longer than expected. Some of the time lost when searching for papers was gained when solving the challenges. Most of these were easier than expected.

H.2 Schedule for core architecture

	Weeks											
Project	2-11	12	13	14	15	16	17	18	19	20	21-29	30
Introduction	■											
Modules		■										
Factory sphere			■	■	■							
Module Communication						■	■					
Configuration manager								■				
Controlling user interface									■	■		
Case study											■	
Correcting thesis documentation												■

Modules
This is used to describe how individual modules should be designed.

Factory sphere
This time is to be used to develop a method for sharing objects between framework modules and external modules.

Module Communication
This time is used to develop a method for communication between framework modules and external modules.

Configuration manager
This time is used to develop a method for controlling configuration files

Controlling user interface
This time is used to develop a method for controlling user interfaces

H.2.1 Actual time used for designing the core architecture

Modules started week 13 27/3 - ended week 13 30/3
The time was used for designing a basic module to use in the framework. One day was gained

Factory sphere started week 13 31/3 - ended week 16 21/4
This module was designed in the limited time allowed.

Module Communication started week 17 24/4- ended week 19 10/5
This module has taken 3 days longer to implement than expected. The main reason for this, it was more difficult to implement the solution than expected

Configuration Manager started week 19 11/5 - ended week 20 17/5
The design and implementation of this module took 1 day less than expected. The reason for this is the implementation was easy.

Controlling user interfaces started week 20 18/5 - ended week 23 9/2
Reducing the application controller pattern and implementing is was more difficult than anticipated. For this reason this module took 5 days more to implement than expected.

H.2.2 Summery

The phase of developing and implementing modules have taken 2 weeks longer than expected. This is mainly because of difficulties regarding implementation of the designed modules.

H.3 Schedule for case study

Before a time schedule can be made, it is necessary to identify the cases that are to be looked at. Each case should be given a case Id.

H.3.1 Defined cases

Case A High speed communication
Purpose of this case is to explore how performance effective communication can be done using the framework.

Case B Configuration security
Purpose of this case is to explore how license relevant information's can be stored in configuration files without being altered.

Case C Distributed system
Purpose of this case is to explore how the system can be split up into several smaller devices.

Case D Interaction between user interfaces
Purpose of this case is to explore how interaction between user interface modules can be performed.

Case E Evolution on communication channels
Purpose of this case is to explore what happens to the communication between modules when the protocol used is evolved.

Case F Controlling measurement location
The purpose of this case is to explore how to ensure measurement modules are displayed at the correct location every time they are displayed.

H.3.2 What to be done during case study

During a case study a small implementation is created to verify the described solution.

H.3.3 Schedule

	Weeks											
Project	2-11	12-20	21	22	23	24	25	26	27	28	29	30
Introduction	■											
Core architecture		■										
Case A			■									
Case B				■								
Case C					■							
Case D						■						
Case E							■					
Case F								■				
Case G									■			
Thesis documentation									■	■	■	
Correcting thesis documentation												■

H.3.4 Actual time used for Case studies

Case A started week 24 12/6 - ended week 15 15/6
This case have been reduced to 4 days to catch up the delays

Case B started week 24 16/6 - ended week 25 21/6
Case C started week 25 22/6 - ended week 26 27/6
Case D started week 26 28/6 - ended week 27 3/7
Case E started week 27 4/7 - ended week 27 6/7
Case F started week 27 7/7 - ended week 28 10/7
Case G started week 28 11/7 - ended week 28 13/7

Thesis documentation and corrections started week 28 14/7 - ended week 31 7/8
This part have mainly been used to collect all the written material and proof-reading of the thesis. Proofreading started 24/7.

H.3.5 Summery

Due to the delays of the previously part of the thesis, the case studies had to be reduced in order to complete them in time.

One week was added to the thesis period due to supervisor's vacation plans. Delivery have been moved from 31/7 to 7/8

Appendix I

Challenges

Purpose of this chapter is to give an overview of problems that needs to be addressed before the design of the application can commence. Each identified challenge will be given an ID for later reference.

I.1 Platform image

Challenge 1

This project requires a Windows CE 5.0 image supporting Compact Framework 2.0 running on the TCM platform. The platform builder contains a limited amount of hardware drivers that can make it difficult to work on the TCM platform. Alternative platforms could be a standard PC image or CE emulator image.

I.1.1 Device image

Before starting configuration of a Windows CE 5.0 image a copy of the current image used on the TCM was analyzed. This was done to have a list of drivers needed for getting the image to work on the TCM device. Using the driver list as a reference several images was created and tested. None of the images tested worked. It was discovered that the driver used to control the graphic card on the device was incompatible with the hardware. This was already known on the old platform, but a small modification to the graphic driver made it work. This modification did however not work on the new graphic driver. It was then decided to find another solution.

I.1.2 PC Image

After 2 configuration attempts a working CEPC image was ready.

Using a CEPC image generated a problem. When calibrating O_2 and CO_2 sensors the current barometer pressure is needed in order to get a precise calibration. Since there is no barometer in a normal PC this value have to be hardcode or typed into the system meaning a precise calibration cannot be made.

During analysis of challenge 3 it was discovered that on a CEPC image COM1 is used to send out debugging information for the image. This was a problem since the only available PC's only had one COM port. Several attempts to disable this feature failed and it was decided to use an emulator image instead. Later it was discovered the reason why the attempts to disable the debugging feature failed. The reason for this is a bug in the boot loader used to initialize the image. The bug and a solution are described at:
http://blogs.msdn.com/mikehall/archive/2005/03/14/395317.aspx
This solution has not been tested since I had a working image for the emulator.

I.1.3 Emulator image

The emulator image was up and running in the fist attempt. Using the emulator give the same problem as using an image on the PC. There is no access to barometer pressure. It also adds the problem of performance testing. Running performance test in the emulator is almost impossible since the system is at the mercy of the host PC and what it is doing at the moment. This means a piece of code can be executed in 1ms the first time executed. The second time it could

take 10 second because the PC is scanning for viruses.

I.1.4 Summery

The emulator image has been chosen as a usable solution. The problem of the barometer pressure can be ignored since this software is not for medical usage. Imprecise value can still be used to demonstrate the basic functionality.

The performance issues using the emulator can largely be ignored since it is not currently known what hardware platform that will be used in the future. However areas where performance could be an issue should be noted for later investigation.

I.2 Connectivity between Visual Studio and CE

Challenge 2

To run and debug applications on the CE platform requires some form for connection between the CE platform and Visual studio.

This challenge could have a high impact on the implemetation schedule since it is an essential part for the success of this project. Searching the help files in Visual Studio 2005, Platform Builder and google search has solved this problem. A small guide for how to get connectivity can be found in Appendix B.

I.2.1 Summery

It seems Visual Studio is mainly designed to create applications for small handheld devices when it comes to compact framework. Support for devices that do not support Active Sync[1] is not directly available. The guide created explains the requirements needed in the CE image and setup procedure required to establish communication. Support for this type of development is better when using platform builder 4.2 and embedded visual studio 4.0.

[1] Active Sync is tool used to establish a connection between a PC and a handheld CE device.

I.3 Serial connection

Challenge 3

The current hardware modules require a serial connection. In Compact Framework 2.0 serial support has been included, but how to use and what limitations the implemented objects have, is currently not known.

The software connection to communicate with modules is an essential part of the project, but the project can be complete using simulated values instead of real measurements from a hardware module.

After several attempts to get serial communication on the CEPC platform, it was discovered the CEPC image was unable to initialize the computers serial port.

The solution to this problem was to make an emulator image and through this communication with serial ports is possible.

The emulators mapping is not entirely logical the emulators COM1 is reserved for serial debugging and cannot be seen from the CE image running in the emulator. The emulators COM2 port is from the CE image seen as COM1. This port can in the emulator's configuration be mapped to a hardware COM port.

Using the serial class in .NET is relative simple.

I.3.1 Summery

Once the image configuration problems were solved communication was immediately available. The serial support in Compact Framwork 2.0 works without any problems.

I.4 Event handling

Challenge 4

.NET has several ways of event handling. These are message queues, delegates and events (Extended version of delegate.). Before designing the application it is needed to know strength and weakness of each event type.

The application will require some type of event handling but how this is done is currently not important and the method should be selected during design of event system.

The message queue system is a new addition to Compact Framework 2.0 and it allows global message queues. This enables cross application communication on the same system and over a network if necessary. It can however be difficult to multicast an event over a message queue since the queue system is designed for multiple senders and one receiver.

The event class is an extended version of a delegate. Both classes contains a list of listener functions that are executed when the event or delegate is invoked. The main difference between events and delegates are, events can be declared in interfaces and there is a restriction on who is allowed to invoke the event. Only the object that owns the event is allowed to invoke it.

Currently the focus of the event system should be on event or delegates. The functionality in the message queue system seems to be more complex than needed.

It should be noted that events and delegates in Compact Framework 2.0 do not support asynchronies calls.

I.4.1 Summery

After implementing prototypes to test the different events methods, they showed the methods are easy to use. Event methods should therefore not be considered as a problem area anymore.

I.5 Software module integrity

Challenge 5

The design is supposed to be modular, with interchangeable modules. Before

initializing a module a check of the module must be performed to ensure it has not been corrupted.

The chances of the disk corrupts a module is low. The current system uses industrial grade flash disks that can handle a large amount write operations before damage occurs. Software modules will properly only be update once or twice per year, meaning the probability for modules being damaged by a corrupt disk is relative small. If however a damaged module is initialized and used, it could have serious impact on application performance. A damaged module could crash the application or give wrong results, causing wrong treatment of a patient.

Compact Framework 2.0 supports strong named assemblies. Strong named assemblies are assemblies that have been signed with a private RSA ref. [10] key and with the public key embedded into the assembly. The RSA key pair can be supplied through a simple key pair file or through a Personal Information Exchange certificate (similar to X.509 certificate ref. [11]). If an assembly is signed then the compact framework will automatically verify the signature. If the verification fails then the loading of the software module will be aborted. If software modules are static linked then all modules have to be signed. Software modules however do not need to use the same key pair. This gives the possibility to have different key pairs to different module suppliers. By creating a list of valid third party public RSA keys and compare these keys to the module key, then it is possible to control supplier's modules. This will however require some form of updating of the key list. Another solution could be that all software modules have to be signed using the same key. This will require some type validation procedure from Radiometer. One problem to consider when selecting a method is that the signing procedure for strong named assemblies is done during build time.

I.5.1 Summery

It is relative simple to create keys and setup the projects to sign the build software. This is no longer considered a problem.

I.6 Third party software modules

Challenge 6

Software modules supplied by a third party must be verified and approved by Radiometer before it is used in a TCM device. This is to ensure these software modules are not malicious or otherwise damaging to the system. The chances of a third party supplier creates malicious software modules are very low but a software module with errors in it could course the application to crash or give bad values. To avoid this there must be a procedure to describe how software modules are tested and a detailed description of software module interfaces.

A solution for module control is described in challenge 5. A test strategy is described in section 1.6

I.6.1 Summery

This is not something that will be looked at further in this thesis. This should only be taken into consideration if the design is to be used commercially.

I.7 Module hardware interface

Challenge 7

Most serial ports on PC use the voltage range -15v to +15v. The measurement modules for the TCM platform use 0v to 5v. To avoid burning out the hardware an interface is required.

I.7.1 Summery

This was discovered not to be a problem at all since such an interface is already available.

I.8 Loading modules Runtime

Challenge 8

In .NET it is possible to load modules runtime using the reflection package. In Compact Framework a reduced version the reflection package is implemented. Some test implementations is required to find out the limits of the Compact Framework reflection package.

A small test application was created (appendix A.1) to demonstrate reflection functionality. This application also makes use of strong named DLL files. A common interface shared among the modules must be designed.

I.8.1 Summery

The test application demonstrated that it is easy to load and execute external DLL files at runtime. If a DLL file is signed the reflection framework will automatically check the integrity of the DLL file before loading it.

I.9 String localization

Challenge 9

If the application is to support multiple languages it is necessary to use multiple resource files containing the applications strings. Since I never have used multiple resource files in a project before, it was necessary to make a small prototype application demonstrating the principle. This should help avoid any problems using resource file during the application design.

I.9.1 Naming resource files

The resource system in .NET has standardized the naming scheme for resource files. Naming resource files in this project have to follow the naming scheme "Module name.LanguageID.resx"

According the resource documentation language ID has to follow ISO 639-1 or ISO 639-2 if ISO 639-1 (ref. [12]) is not possible.

I.9.2 Summery

The test application created (appendix A.1) demonstrates it is possible to create an application containing multiple language files. This makes it possible to switch languages at runtime. The test also showed it can be a little tricky to get it to work. If the language files are not named correctly or if one of the resource files is missing the application could fail unexpectedly. This means great care have to taken to ensure that all the languages exits and they have all the necessary strings. This will require a lot of manual testing.

I.10 Modules version control

Challenge 10

Assembly files can in its standard attributes contain information about the company that have created it and version information. In a standard project these information are located in the "AssemblyInfo.cs" file.

One assembly could be seen as a software module[2]. If this method is used then the standard attributes could be used for version control. This is however not attractive to use since this could result in a very large number of files.

An alternative could be to se an assembly as a package instead. When seen as a package the assembly can contain more than one software module. Since each of the software modules must have the version number it is no longer possible to use the assembly own attributes. These can only be used to version the package. One way to solve the problem could be, by giving each module a GUID value.

A GUID value is a 128bit value normally used to uniquely identify objects. This could be used for version information and also depending on how the value is split, be used to identify the module type. The reason for selecting a GUID value is the possibility extending the information about a module by splitting the value up into smaller parts.

[2]A software module can contain one or more objects

I.10.1 Summery

GUID values should be used to identify modules. The way they should be used must be specified in the design. There is no problem in using GUID values since the amount of unique values should last the lifetime of the application. However if the value is split up into sections some limitations of its lifetime could be introduces. This should be taken into consideration during the design.

Listing the allowed GUID values in a configuration file could be used to control the dependencies between modules and versions.

I.11 Integrity of configuration files

Challenge 11

This system can have multiple configuration files. This could be files containing configuration for a specific module, or files containing descriptions of valid system configurations. To avoid the files becomes corrupted by a bad disk or external editing these files should be signed. In Compact Framework 2.0 digital signing is possible by creating a SHA1 hash value and encrypts it with a private RSA key.

I.11.1 Summery

Securing the integrity is important and if a configuration control system is implemented, the focus should be on the security part.

I.12 Serializing objects

Challenge 12

Serialization of objects can be necessary when saving objects to the disk. This has low priority in the projects since test stubs can be used in these situations.

Compact Framework 2.0 supports serialization to xml files through the "XmlSerializer" objects. This object serializes all public attributes in an object to a stream, as long as these are primitive types. Other types of serialization are currently not supported by the framework. Since it is not attractive to use public attributes in objects and it only supports primitive type, it should be considered to make a custom serialization system. The XML system does however have a method for serializing a XML document to a text files and also to deserialize a text file to a XML document.

I.12.1 Summery

Since serialization is not directly available it should be considered to contain all data that needs to be saved in XML a document. This should make it easier to load and save data.

I.13 FDA and CE approval

Challenge 13

As a medical device the TCM monitor must get approval from several different organisations. Some of these are FDA, UL and CE

To approval from these organisations there is a lot of requirement that have to be fulfilled. To get an overview what these requirements are and if any should be taken into consideration during the system design, a small meeting with Radiometer is needed.

After the meeting about approval, it was clear the requirements are mostly focused on controlling the development process and electrical security. Most of the requirements are therefore not relevant for this thesis, and can largely be ignored. The few requirements that could influence this thesis are the following.

Traceability
: FDA has a requirement about traceability of product requirements in the documentation. The purpose of this traceability seen from a software perspective is the ability to trace a software requirement from requirement description to design and to test. Adding a unique ID to a software requirement can do this.

Test strategy
FDA requires a test strategy, since this is something I need anyway this requirement will also be taken into consideration.

Language localization
CE approval requires a product to be translated into the native language in the target marked. In challenge 9 there is a discussion on how string localization could be done, so this requirements should be taken into consideration during the design process.

I.13.1 Summery

Since the purpose of this project is not to make an application ready for sale, these requirements will have a low priority.

List of Figures

Bibliography

[1] Design Patterns : Elements of Reuseable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
ISBN 0-201-633610-2

[2] Patterns Of Enterprice Application Archiecture
Martin Fowler
ISBN 0-321-12742-0

[3] Evolution of Object Systems or How to tackle the Slippage Problem in object systems
Kai-Uwe M.tzel, Walter Bischofberger
Ubilab, Union Bank of Switzerland
Bahnhofstr. 45, CH-8021 Zurich
e-mail: Kai-Uwe.Maetzel, Walter.Bischofberger@ubs.com

[4] Adaptive Plug-and-Play Components for Evolutionary Software Development
Mira Mezini and Karl Lieberherr
College of Computer Science, Northeastern University, Boston, MA 02115-9959
E-mail: fmira,lieberg@ccs.neu.edu

[5] Composite Design Patterns
Dirk Riehle
Ubilab, Union Bank of Switzerland, Bahnhofstrasse 45, CH-8021 Zurich
Phone: +41-1-234-2702, fax: +41-1-236-4671
E-mail: Dirk.Riehle@ubs.com or riehle@acm.org

[6] A Layered Architecture for Uniform Version Management
Bernhard Westfechtel, Bjùrn P. Munch, and Reidar Conradi, Member, IEEE

[7] A Simple and Practical Approach to Unit Testing : The JML and JUnit Way
Yoonsik Cheon and Gary T. Leavens
ECOOP 202 - Object-Oriented Programming
ISBN 3-540-43759-2

[8] Patterns as Signs
James Noble and Robert Biddle
ECOOP 202 - Object-Oriented Programming
ISBN 3-540-43759-2

[9] http://en.wikipedia.org/wiki/Design_pattern_(computer_science)
Wiki containing links to different design pattern sites.

[10] http://en.wikipedia.org/wiki/RSA
Information regarding RSA encryption usage.

[11] http://en.wikipedia.org/wiki/X.509
Description of the X.509 certificate.

[12] http://www.loc.gov/standards/iso639-2/englangn.html
Link to ISO 639 1 and 2.

[13] http://en.wikipedia.org/wiki/SHA
Link to SHA algorithms.