Documenting and Improving the Design of a Large-scale System

Gabriel Toresson

Supervisor: Jonas Wallgren
Examiner: Martin Sjölund

External supervisor: Per-Ola Lindell
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Gabriel Toresson
Abstract

As software systems become increasingly larger and more complex, the need to make them easily maintained increases, as large systems are expected to last for many years. It has been estimated that system maintenance is a large part of many IT-departments’ software development costs. In order to design a complex system to be maintainable it is necessary to introduce structure, often as models in the form of a system architecture and a system design. As development of complex large-scale systems progresses over time, the models may need to be reconstructed. Perhaps because development may have diverted from the initial plan, or because changes had to be made during implementation. This thesis presents a reconstructed documentation of a complex large-scale system, as well as suggestions for how to improve the existing design based on identified needs and insufficiencies. The work was performed primarily using a qualitative manual code review approach of the source code, and the proposal was generated iteratively. The proposed design was evaluated and it was concluded that it does address the needs and insufficiencies, and that it can be realistically implemented.
Acknowledgement

I would like to thank my supervisor at Saab Aeronautics, Per-Ola Lindell, for all the excellent help and support, it has been greatly appreciated. I would also like to thank Anders Bredberg at Saab Aeronautics for giving me the opportunity to perform this thesis.

Additionally, I would like to thank my supervisor Jonas Wallgren and my examiner Martin Sjölund for all the support and constructive feedback, also greatly appreciated.

Gabriel Toresson
Linköping, Sweden
June 2019
List of Figures

Figure 1. Example of a Class diagram (left) and a Sequence diagram (right) (adapted from [7]) ........... 3
Figure 2. Custom symbols for Qt connects ................................................................................................ 4
Figure 3. Illustration of the Facade design pattern (adapted from [7]) .................................................. 7
Figure 4. Illustration of the Observer design pattern (adapted from [7]) .............................................. 7
Figure 5. Informal application overview ................................................................................................. 13
Figure 6. Informal view of contents and relations of the primary Intplot3 classes .............................. 14
Figure 7. UML class diagram of the primary Intplot3 classes ................................................................. 15
Figure 8. Diagram of Intplot3 start-up procedure ................................................................................... 16
Figure 9. Parts of the diagram of the MainWindow creation ................................................................. 17
Figure 10. Parts of the diagram of the GLWidget creation ..................................................................... 18
Figure 11. Diagram of the autoUpdateGL() function .......................................................................... 19
Figure 12. Diagram of the creation of AppCore .................................................................................... 20
Figure 13. Diagram of the function PlotManager::Init() ..................................................................... 20
Figure 14. Part of the diagram of the function GLWidget::paintGL() .................................................... 21
Figure 15. Diagram of the function GLWidget::renderOnCurrentContext() ........................................ 22
Figure 16. Diagram of the function PlotManager::Plot() ..................................................................... 23
Figure 17. Diagram of the function PlotFaceOGL() ............................................................................. 24
Figure 18. Diagram of the function PlotFace::PlotFace() .................................................................... 25
Figure 19. High-level diagram of the proposed system design .............................................................. 26
Figure 20. Class diagram of the CanvasComponent .............................................................................. 26
Figure 21. Class diagram of the CoreComponent .................................................................................. 27
Figure 22. Low-level diagram of the RenderComponent ....................................................................... 27
Figure 23. Pseudo-code example of RenderManager::Paint() ............................................................ 29
1 Introduction

Over time, as software systems have grown and become increasingly complex, the need for structure has become ever more prominent. The sheer scale of many of today’s systems means that tools and methodologies are often required in order for a system to be developed in an effective and manageable way. The builders of these systems are often several developers, or even multiple cross-functional teams, all of who need to share the same vision for what the end product will be. As development progresses and the system grows, the initial vision may naturally fade. Unforeseen issues and fixes, parts not realized as planned, changes to the requirements; the structure may have become hard to discern. If the vision of what and where to implement is obscure, the risk of, amongst other things, duplicating functionality and introducing bugs increases. Situations may arise where the developers have to perform quick-fixes, which may further impede future development and leave much to be desired for when it comes to later maintaining the code base.

Large-scale software carries a large inherent monetary value, if anything they are the product of potentially thousands of man-hours; these systems are expected to last for many years or often decades. Given this evolution of software development, the importance of maintainability has escalated rapidly. In 1991 it was estimated that 50-80 % of software development costs of IT-departments were dedicated to maintaining software [1]. Ten years later in 2001 software maintenance was still estimated to stand for 40-80 % (60 % on average) of software development costs [2]. The objective of building a system that is easy to maintain is therefore an important one, if only for reducing development costs. If the longevity of the software system is key, then the maintenance cost will potentially outgrow the cost of the actual development over the system’s lifespan. Also, for logical reasons, a system that is hard to maintain can also be difficult to develop further, for example due to obscurity or risk of errors.

Structure to manage scale and complexity comes in different forms, it can for example be through following naming conventions, using development methodologies, or developing a system architecture and a system design. The two latter ones can be crucial for developing large-scale applications as they make sure that the system can meet its requirements. They also unify the developers on a single system model. As such, a system architecture and a system design can greatly reduce development risks which could otherwise cause significant problems later on in the development process.

1.1 Background

Operating on the global market, Saab is a company which provides military defense and civil security products and solutions to various authorities, corporations and governments worldwide. Their selection of products and solutions ranges between naval, land and aviation applications.

The business area Aeronautics focuses on development of aeronautical products with military and civilian applications. During product development numerous simulations are performed regarding stress engineering, for example using the Finite Element Method (FEM). The results of these calculations can then, together with the finite element model, be visualized in a software such as Intplot3, a proprietary tool developed at Saab in Linköping. Intplot3 is the focus of this thesis. The models and FEM-results are stored in a format called IntSyS.

Intplot3 uses a total of five libraries in order to function: Intsys, Intsyspp, Iparser, Gtoolsqt and MathLib. Intsys parses IntSyS-files. Intsyspp constructs data models of the parsed finite element models and FEM-results. Iparser parses user input. Gtoolsqt and MathLib are two libraries which provide rendering functionality and mathematical functions respectively. The application can be run in batch or graphical user interface (GUI) mode.
Intplot3 is an extensive and complex software system whose purpose was to modernize a previous application (Intplot2) written in FORTRAN. Intplot3 was meant to be based on the object-oriented programming paradigm and introduce interactive three-dimensional graphics. Given the size and complexity of Intplot2 it could not be quickly replaced, and as a result Intplot3 was built on top of it. Old functionality is being incrementally ported to Intplot3. The main functionality of the application is contained in Intplot3 as well as in the Intsyspp library, both written in C++. The total size of the application is large. Intplot3 alone is made up of over 400 cpp and header files, equivalent of almost 88’000 lines of code. The old functionality of Intplot2 is contained in over 1100 FORTRAN files, equivalent of over 175’000 lines of complex code. Intsyspp is roughly of the same size as the modern part of Intplot3.

The application is based on the concept of commands which the user can enter in order to load files, calculate results, generate graphics and so on. If Intplot3 is run in GUI mode then the commands can be sent either manually or using menus and dialogs. The way these commands are passed through the application is called the command flow, it involves receiving, parsing and executing appropriate functionality.

The GUI in Intplot3 uses the event-driven framework Qt. The modern rendering uses OpenGL to render the models, but the old code can still be used to render them as well. Version 4.8 of Qt and version 4.6.0 of OpenGL are used in Intplot3.

1.2 Motivation

The rate of development of Intplot3 has been high, with several temporary resources. This has over the years led to a shift of focus away from system architecture and system design. Parts of the current system structure and interactions therein are therefore today perceived as somewhat obscure, making the code difficult to maintain and develop further. In particular, the modern rendering functionality in Intplot3 is in need of attention.

1.3 Aim

The aim of this thesis is to first create a model of the existing system architecture and design and then generate suggestions for improvements. The model and the suggestions will be focused primarily on the modern rendering functionality of Intplot3. These can then be used to guide the future development process. The suggestions will be based on identified high-level needs and insufficiencies which exist in the system today.

1.4 Research questions

- What does the existing system architecture and design look like at a high level?
- What needs and insufficiencies exist for, and in, the existing system?
- How can the system architecture and design be improved?

1.5 Delimitations

Given the large scale of the application, this thesis will focus on the modern rendering in Intplot3. The models of the existing system and the final proposal will be on a relatively high level, with some simplifications made, as to not go out of scope of the thesis. Since much of the work is done manually, without automated tools, the granularity of the documented system model will be limited to provide an overview of the important structures and interactions within Intplot3. The suggested improvements will be a high-level system design, no code will be rewritten or implemented. The number of identified needs and insufficiencies will be limited to the most important ones.
2 Theory

The following chapter presents the theory which is the basis for the existing system model, the proposed system design and its evaluation. First the technical concepts and frameworks are introduced, followed by the more theoretical concepts.

2.1 UML

The Unified Modeling Language (UML) is a general-purpose modeling language regarded as the standard for modeling software systems [3]. UML 2.0 is a major revision published in 2005; it was released as two complementary specifications: Infrastructure [4] and Superstructure [5]. The Infrastructure specification was the first of the two to be released; it defines the foundation of the language. The Superstructure specification is focused on defining constructs at the user level.

As a formal language, UML allows for concise yet accurate modeling of both small- and large-scale systems. From a developer’s perspective, modeling a system is a way to express it in a more abstract and manageable way, all to manage complexity. The intent of UML has been to standardize the visualization of system design, so that it can be communicated effectively and without error between multiple developers and stakeholders. It is neither based on a single programming language, nor is it dependent on the application’s environment. While UML is mainly used in software development, it can be used to model non-software systems as well, such as in business modeling [6].

Using UML, a system can be expressed as a model, which is an abstraction of the actual implementation. It is not meant to be a 1:1 representation, instead, the model exposes important aspects of the system using a set of views, often in the form of diagrams. Based on what is relevant for the task at hand, views can be very different from other views, yet they can relate to the same part of a system.

Two common diagram types are class diagrams and sequence diagrams. Class diagrams are used to represent static structures of classes in a system, and sequence diagrams depicts dynamic interactions (in other words interactions during runtime). The class diagrams are concerned with logical relationships between classes, while sequence diagrams are concerned with where and when interactions take place [3]. For this reason, sequence diagrams are often related to a specific scenario or process and are typically not used to model the entirety of a system. Figure 1 illustrates examples of one class diagram and one sequence diagram, respectively.

![Figure 1. Example of a Class diagram (left) and a Sequence diagram (right) (adapted from [7])](image)

2.2 The Qt framework

Qt is a cross-platform C++ framework often used for developing applications with a GUI [8]. It can be extended, using modules, to support OpenGL rendering. The GUI can be created using widget components, for example windows, menus, dialogs and buttons, which can react to user interaction using an event-based implementation. Throughout this work, classes whose name contains a Q prefix, such as QApplication, are Qt classes.
QGLWidget is a Qt class that exposes OpenGL rendering functionality inside a Qt application. It provides three key virtual functions that can be reimplemented by a developer: an initializer (initializeGL()), a paint-event handler (paintGL()) and a resize-event handler (resizeGL()). The initializer sets up OpenGL to be used in the widget. When Qt wants to trigger a repaint or resize of the widget, then paintGL() or resizeGL() will be called, respectively. If an exterior (as in not Qt) repaint should be triggered, for example during animations, then the function updateGL() should be called [9].

The recommended way to render three-dimensional graphics using OpenGL in conjunction with Qt is to create a QGLWidget and then reimplement all three previously mentioned virtual functions. The OpenGL state should be initialized or updated in initializeGL() or resizeGL() respectively. The function paintGL() should then perform all OpenGL calls needed for drawing the object onto the widget [9].

An important feature in Qt is the so-called signals and slots [10]. At a conceptual level they resemble events; a class can emit a signal which calls a receiving slot in another class. These signal-to-slot associations are set up using connects that specify the classes and which signal goes to a specific slot. The signals and slots can be within a single class or between two different classes. One-to-many signals are set up using multiple connect statements. This allows for a sender to not be aware of the receiver, which reduces coupling.

In UML there is no pre-defined way of representing connects, so the following symbols have been introduced (see Figure 2): The text “<<connect>>” written next to a diamond shows when and where a connect statement is executed. The arrow represents a signal that goes to a slot. The text above the arrow is the emitted signal’s name. For example, the first scenario in Figure 2 shows a connect statement executed in ClassA, it registers a signal by the name of changeCmd() that goes from ClassB to a slot in ClassC. The last scenario shows a connect executed in ClassA that registers a signal that goes to a slot in the same class.

![Figure 2. Custom symbols for Qt connects](image)

2.3 OpenGL

The Open Graphics Library, or OpenGL, is a cross-platform API for two- and three-dimensional graphics, regarded as the industry standard [11]. It is being used extensively in a wide variety of applications on many different platforms. Unlike what the name suggests, OpenGL is not a library per say, it is a specification which is then implemented in the graphics drivers. Despite being a specification, versions of OpenGL are still partly related to graphics hardware tiers [12].

In 2008 version 3.0 was released and it brought with it a deprecation model which specified much of the earlier functionality as deprecated, now often referred to as legacy OpenGL. Version 3.1 removed most of the deprecated functionality, but in order for old code to keep working even for newer versions the concept of profiles were introduced. In version 3.2 OpenGL was split into two profiles, core and compatibility, and the deprecated functionality...
was only removed from the core profile. Version 3.1 introduced a similar concept, but it was not sufficient enough and it was therefore replaced by the Core and Compatibility profiles [13].

As a specification, OpenGL only requires implementations to support the core profile and as such, support for the compatibility profile may vary [13]. Therefore, using legacy code can introduce compatibility issues as the code will only run if the graphics device is backwards-compatible with the legacy versions. Should the manufacturer decide to remove the compatibility profile in new drivers then the code will also cease to work. While there is still wide support amongst manufacturers for the compatibility profile, there are manufacturers who have abandoned such plans for certain operating systems [12] [13].

Functionally, OpenGL is based on the concept of a pipeline called the rendering pipeline. It is a sequence of steps which is traversed by OpenGL when rendering an object using the GPU, steps such as vertex processing, interpolation and rasterization [14]. In legacy OpenGL there is a concept called fixed function pipeline, where the code basically uses specified matrices, settings and OpenGL calls to generate an output. Many of these features were replaced by shaders in the newer versions of OpenGL [15]. Shaders are user-defined programs running on the GPU, overriding some stage in the rendering pipeline [16].

OpenGL calls, just like normal function calls, are blocking operations, the rest of the code must wait until the CPU finishes a call. Executing a large number of calls can affect performance as the CPU is too busy to process other things (like the GUI). Thus, reducing the number of unnecessary OpenGL calls and state-changes can improve performance. Shaders can reduce the number of calls even further as much of the data can be buffered on the GPU, data such as vertices for a group of elements.

2.4 Software quality attributes

Over the years the need for software quality has increased as applications have grown drastically in terms of both size and complexity. To meet this demand, developers require some form of software quality metrics that can be used to identify problems and guide the development process.

In order to measure any type of quality it must first be known what is considered “quality” in a particular scenario, and what characteristics does it have that can be measured? In order to standardize software quality evaluation, the International Organization for Standardization (ISO) published the ISO/IEC 9126-1:2001 standard [17] (henceforth called ISO/IEC 9126). It classifies software quality as a set of characteristics, each with a set of sub-characteristics. The standard defines the characteristics “Functionality”, “Reliability”, “Usability”, “Efficiency”, “Maintainability” and “Portability”. The characteristics can then be divided into sub-characteristics that can be measured in a system, given a context and an appropriate metric. The ISO/IEC 9126 standard has since been replaced by ISO/IEC 25010:2011 (henceforth referred to as ISO/IEC 25010), a standard that restructures parts of the previous quality model, as well as adds and renames several characteristics and sub-characteristics [18]. Based on the updated quality model, two characteristics are relevant for this work: “Maintainability” and “Performance efficiency”.

Maintainability warrants further explanation; it is described as “degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers” [18]. There are three relevant sub-characteristics of Maintainability: Analysability, Modifiability and Reusability. Quality characteristics and sub-characteristics are often called Quality attributes, throughout the rest of this work they will be referred to as such.

In order to measure the software quality attributes one or several appropriate metrics must be selected for use. A number of quality models and metrics have been proposed over the years, a known one is the Quality Model for Object-Oriented Design (QMOOD), developed by Bansiya and Davis [19]. Their work is aimed at developing a model and related metrics which
can evaluate software using mainly design information. QMOOD defines six quality attributes, three of which are Reusability, Flexibility and Understandability. QMOOD is based on earlier models and the attributes defined in the ISO/IEC 9126 standard. After reviewing object-oriented development books and publications, Bansiya and Davis propose a set of design properties, such as design size, coupling and encapsulation, which they state directly affects a number of their quality attributes. To measure these design properties they propose a set of metrics designed to be usable using design information.

Chidamber and Kemerer [20] proposed another well-known metrics suite, often referred to as the CK metrics, consisting of six metrics, where only three can be applied at the design level.

In a survey of software quality metrics applicable to UML class diagrams [21] a wide range of metrics were reviewed, among them were QMOOD and the CK metrics.

Based on the survey and the scope of this thesis, no metrics are fully applicable to this work. The generated UML views of the proposed system design are on a too high level, there are many simplifications and there is no implementation to measure.

Comparing the definitions of the QMOOD attributes to the attributes defined in the ISO/IEC 25010 several similarities can be observed. Reusability as defined by both models are essentially the same. Modifiability of the ISO standard compared to QMOOD’s Flexibility are also similar. QMOOD’s Understandability however is more abstract than Analysability of the ISO standard, but they do share similarities when it comes to comprehending a system. Bansiya and Davis state that the Maintainability defined in ISO/IEC 9126 was replaced with QMOOD’s Understandability in order to focus on the design characteristics of a system. Their view was that Maintainability implied that a system already existed, and that the attribute was therefore not suitable for their proposed model. This does mean that there are similarities at the abstract level between Analysability of ISO/IEC 25010 and Understandability of QMOOD.

2.5 SOLID design principles

SOLID stands for a set of five design principles for object-oriented programs. Their goal is to make software structures that are easy to understand, that tolerate change and that are the basis of components that can be reused [22]. While the principles are aimed at object-oriented systems, they can be used in other software systems as well. The five design principles that make up SOLID are:

- Single responsibility principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

For this work the Single responsibility principle is important, it can be described as: a class should only have one responsibility; it should only have one reason to change [22].

2.6 Design patterns

The book Design Patterns: Elements of Reusable Object-Oriented Software [7], often denoted Gang of Four or GoF, is by many regarded as the standard for design patterns. Detailing 23 different patterns, the book essentially aims to provide standardized solutions to common programming problems. Each pattern is part of one of three groups called Creational Patterns, Structural Patterns and Behavioral Patterns, named after what they are designed to impact. The benefits of the usage of design patterns need to be further researched as concluded by a systematic literature review by Zhang and Budgen [23]. They found that there was support for
that design patterns improves communication between developers, but that many other claims were unsupported, often due to poor research documentation. The individual design patterns themselves, however, are still perfectly valid solutions to many types of common problems, and as such they may still be implemented, or used as a base for a custom implementation.

2.6.1 Facade
The *Facade* pattern is categorized as a Structural pattern. Its intent is to “provide a unified interface to a set of interfaces in a subsystem that makes the subsystem easier to use” [7]. Commonly used in component-based systems (discussed in 2.7.1), the facade class acts as the interface to most of, or even the entire, component. By reducing the number of interdependencies (coupling) within a system the overall quality of the code base can be increased, as coupling negatively affects several system qualities such as reusability and extendibility [19]. The facade class can also increase flexibility within the application. Given that it separates a subsystem by a unified extra level of abstraction, the subsystem behind the facade could potentially be interchangeable. Figure 3 illustrates the Facade pattern.

![Figure 3. Illustration of the Facade design pattern (adapted from [7])](image)

2.6.2 Observer
The design pattern *Observer* is part of the Behavioral category. Sometimes called Publish-Subscribe, the Observer pattern uses an event-based design to define one-to-many communications between objects. Events that affect one object, such as a state change, are automatically propagated to all its listeners. The motivation behind the pattern is primarily to be able to maintain consistency between related objects, without making them tightly coupled in the process [7]. The object broadcasting to all listeners do not need to be aware of what or how many objects are listening, which allows for highly dynamic communication within a system. Figure 4 illustrates the pattern; ClassA updates ClassB which in turn broadcasts to all listeners (observers) that its state has been updated.

![Figure 4. Illustration of the Observer design pattern (adapted from [7])](image)
2.6.3 Anti-patterns

The term anti-pattern is similar to the concept of design patterns, except that instead of offering an improvement, in reality it may be ineffective or even harmful [24]. Anti-patterns exist in many different fields and at different levels. A common one in software development is the God class, also known as the God object. It is a class/object that essentially does too much, it is too broad in terms of functionality and responsibilities which often makes it hard to manage and understand. Typically, the God class is considered bad practice, especially in large-scale systems where said class can become large and obscure.

2.7 Software architecture and design

Complex software can be said to be composed of three principal layers, from the top down the system has an architecture, a design and at the bottom an implementation. As with any type of construction, it is important that the foundation is solid and appropriate for what the end product will be. A commonly quoted definition on the concept of software architecture is:

“The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships between them.” [25]

That is, a system’s software architecture can be said to be its skeleton, the highest level which define the overall structure and the interactions between the system’s components, without specifying how things are done. Quality attributes can be applied at different levels throughout design process, several of which can be designed for in the architecture in such a way that allows the end product to meet expectations.

The divider between software architecture and software design is not always easy to distinguish. While the architecture is the skeleton, the design is essentially about responsibilities and functions of components, or even individual classes, and the relationships between them. The software design can also be said to be where many problems are conceptually solved, as well as where design patterns are applied. It can therefore be said that the fundamental difference between the architecture and the design is the scale. Both the architecture and the design incorporate the software quality attributes, which is important as the final implementation builds on the design.

Software architecture and design are often modeled using a modeling language such as UML, and the model often consists of class diagrams, sequence diagrams, flowcharts and so on using different views.

2.7.1 Component-based systems

When designing an architecture it is common, at a high level, to separate responsibilities into logical components, where each component has a single, clearly defined, purpose. Not only is the structure easier to visualize. Encapsulating related data and functionality means that the components can in principle be reused and interchangeable, since tight coupling can be avoided. For example, a webserver application may have a database component, an authorization component, a view component and so on, each with a single responsibility. Given that components are characterized as reusable and interchangeable, as such they communicate using interfaces to achieve the loose coupling. Each component is a sub-system responsible for performing a specific task.
2.7.2 Event-driven architecture

A component-based design can be used in conjunction with an event-driven architectural style, a style which promotes interactions based on the production of, and reaction to, events. Components can be producers and receivers of events. In its simplest form, an event signifies a significant state change within a system which is then propagated to all related objects that are awaiting that specific change. For example, user input in a GUI will trigger events such as mouse movement events, button press events and so on, but events can also be used internally to propagate changes to related objects. The main benefit of an event-driven style, other than the system being able to handle exterior influences and asynchronous functionality, is that it promotes loose coupling between transmitting objects and receiving objects. It also makes the system scalable as event-driven design is not bound by hierarchies: any object can produce events, and any object can listen to them.
3 Method

This chapter presents the procedures used during the various stages of this thesis. The work was separated into three stages in order to answer the three research questions. These stages were: documenting the existing system, identifying needs and insufficiencies, and developing an improved system design. The stages are listed in chronological order in which they were performed.

3.1 Documenting the existing system

The existing system was documented using UML class diagrams to present the overall structure and sequence diagrams to map the internal interactions. The interactions during two key application phases was mapped, those two were:

1. Intplot3 start-up
2. The main rendering loop

These two selections were made as they cover a wide range of important internal interactions within Intplot3.

Generating UML class diagrams is something that can be automated using a static code analysis tool, that is, a tool which parses and analyses a system’s static structure (classes), mostly using the source code. Due to a secure computer environment at Saab the availability of such tools was limited (discussed further in 5.2). The most suitable tool available was the software Enterprise Architect, a modeling tool compatible with UML. It was used to parse Intplot3’s header files and subsequently output class diagrams with dependencies. Intplot3 uses mainly C++ code but there is also C code mixed in with interfaces for FORTRAN functions. This mix, coupled with the fact that includes can be specified in both the header and cpp files, resulted in the automatically generated output being incomplete. Dependencies as well as classes were sometimes missing from diagrams. The results were therefore not reliable enough to use as documentation, or to base any proposals on. While the problems could potentially be fixed manually, the results would still not be fully reliable. If a few out of roughly 200 header-files were missing it may not be noticeable in the generated diagrams, meaning that future work could be based on incomplete information. As such, the decision was made to use a systematic manual code review approach. This would have been necessary regardless as the tool (given that it could not parse cpp files) could not construct sequence diagrams automatically. The result generated by Enterprise Architect was instead used as an outline that aided the code review.

The manual code review started with the first application phase, the start-up. The start of the application is clearly defined in C++, it starts at the `main()` function. The review went through the code line by line until the end of the call structure had been reached, which also signified the end of the first phase. All connected classes and functions were noted as they were the basis for the class and sequence diagrams that were later constructed (presented in 4.1.1).

Documentation of the second phase was performed using the same approach, going through the code line by line. The second phase is initiated during the first phase, as part of the start-up, which means that there was a known starting point for the second code review as well. All connected classes and functions relating to the second phase were also noted and later used to construct a set of sequence diagrams (presented in 4.1.2).

After both phases had been mapped a set of core classes had been identified. All includes of these classes were categorized as to be able to provide a high-level overview of each of their responsibilities and functionalities (presented in 4.1.1).

Both the UML class diagrams and sequence diagrams were constructed mainly based on the findings of the code reviews and the generated outline. The review covered both the cpp and
header files. As sequence diagrams can visualize all types of interactions, both programmed (function calls) and dynamically generated (events), it was necessary to complement the code review. This was done by analyzing the patterns of interactions during runtime (called dynamic code analysis). At the same time, this also allowed for the cross-checking of the mapped call structures to compare what was meant to happen to what did happen.

3.2 Identifying needs and insufficiencies

Based on the results from the manual code review processes a list of issues and bottlenecks had been identified. In collaboration with the supervisor at Saab, this list was formulated as a set of needs and insufficiencies that should be addressed by the new system design. The needs were then associated with software quality attributes, discussed in 2.4, by comparing the needs to the definitions of the attributes. The needs, insufficiencies and quality attributes (listed in 4.2) were used as the basis for the development of the improved design.

3.3 Developing the improved system design

The work on developing a proposal for an improved system design, more specifically focused on the model rendering in Intplot3, was based on the generated system documentation, the list of identified needs and insufficiencies as well as the associated quality attributes. The final proposal had to be realistic and based on the structure of the system today, given the size of Intplot3 and that there were no large risks to warrant a large redesign. As such, the proposal on an architectural level only sought to improve on already existing styles.

During development, all classes and functionalities relevant to the focus of the thesis was grouped up. All of them were reviewed manually in order to categorize them into different levels of abstraction and different types of functionality. With this knowledge, a new system design could be developed that would address the identified needs and insufficiencies.

The development of the proposal was done in an iterative manner. Findings, issues and potential solutions were discussed continually together with the supervisor at Saab, allowing the design to be altered if necessary. The iterative process also ensured that the final design proposal was viable from their point of view. The results are presented in 4.3.
4 Result

The results from the system documentation, identification of needs and insufficiencies, as well as the development of the improved system design are presented in this chapter.

4.1 Documentation of Intplot3

The documentation has in this section been divided into the two important phases mentioned in 3.1: the start-up and the main rendering loop. The first documented phase will also present the relevant high-level application and class structures of Intplot3.

4.1.1 Intplot3 start-up

A component-based architectural style can be vaguely distinguished underlying Intplot3, as well as an event-driven style given the nature of the Qt framework. All functionality inside Intplot3 can be separated into one of two main categories, UI and Logic. As their names suggests, UI consists of classes relating to Qt and the GUI (such as windows, widgets and rendering). The Logic category consists of classes which provide the core functionalities of Intplot3, such as command parsing and interacting with the finite element models.

At the highest level of abstraction, Figure 5 shows an informal overview of the whole Intplot3 application and its two main categories. The application contains five other linked libraries on which Intplot3 is directly or indirectly dependent as described in 1.1.

Counting the Main class of Intplot3, which starts the application in batch or GUI mode, there are a total of six identified primary classes of importance in Intplot3:

- **Main**, runs the application
- **MainWindow**, the container for the entire GUI
- **IntplotCoreDelegate**, handles high-level interactions, head of the command flow
- **GLWidget**, the “canvas” of the custom rendering, handles user interaction
- **AppCore**, application logic, manages the old core
- **PlotManager**, handles the custom rendering of elements and animations

The classes Main, IntplotCoreDelegate and AppCore belong to the Logic category, while the other three belong to the UI category. Henceforth referred to as the primary classes, their contents were categorized by their includes’ high-level responsibilities and then their relationships were mapped. This is depicted on an informal and simplified level in Figure 6. Note that the class AppCore is split into two cpp files due to its size and wide functionality: **AppCore** and **AppCore_CmdCallback**. For all intents and purposes the two can be regarded as separate classes, which is the reason for the split shown in the figure. The two “classes” are presented in

![Figure 5. Informal application overview](image-url)
Figure 6 as having somewhat differing contents. This is to capture the property of C++ where includes can be specified in the header file or in any of the class’s cpp files.

The categories shown inside each class in Figure 6 are the result of the code review of each of the primary classes. This categorization is (mostly) not mirrored in the Intplot3’s folder structure, it is only intended to provide a rough overview of what other classes are included, which serves to indicate what each class’s responsibilities are.

Some categories are self-explanatory, but some require explanations. Core represents either some of the primary classes, or classes of high significance to the class’s operations; IntplotCoreDelegate is for example categorized as Core in MainWindow. Output represents print functionality, like logging of commands and errors. Misc covers a range of categories for classes such as custom string and matrix data types, classes providing general functionality or simply classes whose category are irrelevant for this view. Data model represents the custom data structures that contain the loaded model files (explained more in depth later). The Rendering category contains classes which are related to the custom OpenGL rendering. In the class PlotManager we see two related categories, Rendering and Plotting. The difference between them is that Plotting classes are on a higher level of abstraction, they mostly handle model elements. This distinction, however, is only valid at the informal level. Throughout the code base, and in
the coming diagrams, these two terms are used interchangeably as a general term for any kind of custom rendering.

All categories shown, including Iparser in AppCore, are related to classes only. Dependencies to Intsyspp (and the other linked libraries shown in Figure 5) can be found in many classes throughout Intplot3, but they were not added to Figure 6 as their own categories. Such classes were instead added to the other categories. Iparser, however, was deemed important enough to warrant its own category as it directly affect the command flow of Intplot3.

Figure 7 presents Figure 6 as a formal UML class diagram. Due to the sizes of the classes, function names are not shown, neither are but a few member variables. In this diagram AppCore and AppCore_CmdCallback are one class, as they are in the code. We also see that GLWidget is contained inside another Qt widget (for layout purposes) and it also inherits the noteworthy `GrDeviceInterface`. The underlying purpose of GLWidget is to act as a canvas that is to be drawn on, and because of that it is meant to be interchangeable which warrants a common interface. What is not shown in the diagram is that GLWidget also inherits the QGLWidget, described in section 2.2. Figure 7 shows how AppCore operates on GLWidget through the `GrDeviceInterface`. The diagram also shows that the relation between AppCore and GLWidget to PlotManager is of the weak type `dependency`. This is because PlotManager is a “static class”, its functions and member variables are static. From here on, when a class is said to be static this is what is referred to. Lastly, looking at the member variables in Figure 7, we note that there is no naming convention used. All shown member variables are pointers, yet the prefixes differ.

As previously mentioned, Intplot3 can be run in batch or GUI mode. Regardless of choice the Qt framework is used, only that the MainWindow is hidden if Intplot3 is run in batch mode. Figure 8 shows the start-up procedure of the application. An actor enters a command from the terminal which ultimately results in the creation of a `QApplication`, which in turn is linked to
the newly created MainWindow. Based on the specified mode, the MainWindow is either hidden or made visible. If the user has specified an input file (for example a model) then Main will send an appropriate command to IntplotCoreDelegate to trigger the loading of the file. IntplotCoreDelegate is the start of the command processing; commands arrive either from direct function calls or from connected events. IntplotCoreDelegate will forward the commands that have arrived to AppCore for parsing and handling.

![Diagram of Intplot3 start-up procedure](image)

Figure 8. Diagram of Intplot3 start-up procedure

If Intplot3 is run in GUI mode then, when the GUI is ready to be started, the `exec()` function is called on the created QApplication. This starts the Qt main event loop, which is required for the GUI to function, and it returns when the user quits the application.

The MainWindow class is the parent of all GUI components, it creates all widgets and sets up the connections between them as depicted in Figure 9. The full diagram can be seen in Appendix A. There are five important connections made, all connecting GLWidget and IntplotCoreDelegate to each other and to other classes. The signal resized(), originating from GLWidget, is important for understanding internal communications. When Qt detects that the
user is resizing the window, or a widget, then a series of resize events are passed down the hierarchy to all affected widgets. When (if) a GLWidget instance gets a resize-event then its new settings need to be propagated throughout Intplot3, as much functionality is spread out into separate classes. This propagation starts with the mentioned signal.

Continuing down the primary class hierarchy, Figure 10 presents part of the creation of GLWidget. The full diagram can be seen in Appendix B. A relatively simple initiation, however, the `autoUpdateGL()` function call is very important as it is what starts the GLWidget main rendering loop, which is shown separately.
Figure 10. Parts of the diagram of the GLWidget creation

Figure 11 presents the sequence diagram of the autoUpdateGL() call referred to in Figure 10. Note the outer loop-fragment, highlighted in its condition is a time interval. The loop repeats every $X$ milliseconds and it is set using the `SetSingleShot(X ms)` at the very end of the activation bar of GLWidget. It is a Qt `QTimer` function (its invocation has been simplified in the diagram), it will invoke autoUpdateGL() after a set amount of time. The first call is done at the creation of GLWidget shown in Figure 10, afterwards it will repeat using the "shots". The rendering loop will repeat with an interval of 100 frames per second (FPS) until one second has passed since the last newly rendered frame, at that point the shots will repeat at 20 FPS.

Another point of interest is the mid-level opt-fragment, if GLWidget is not being painted on, this is what forwards the render call down the hierarchy using `updateGL()`. This function call will cause Qt to schedule a repaint-event of the object in question, here GLWidget. Inside the same fragment there is an `Animate()` call to Intsyspp, seemingly without any conditional attached to it. This is a call to `dead code`, code whose result is no longer used but has not been removed.
The class IntplotCoreDelegate is one of the primary classes, but its creation is not significant (it only creates AppCore), and as such it is being left out of this chapter. The diagram can however be found in Appendix C.

The MainWindow class creates IntplotCoreDelegate, which as mentioned creates AppCore, whose creation is what is shown in Figure 12. As stated in 1.1, Intplot3 is built on top of old FORTRAN code. This old code is denoted the “old core”, and AppCore which represents the modern core of Intplot3 manages the old one through a set of interfaces. The initiation of the old core is shown in Figure 12, initOldCore(). The diagram also shows the creation of the parsing utilities of Iparser. AppCore communicates with Iparser (CmdParser) through an invocation and a callback function. When a command has been registered it is sent by AppCore to CmdParser, and when the command has been parsed the result is returned by invoking the CallbackWrapper function. The result of the parsed command is then forwarded to functionality residing inside AppCore_CmdCallback, which executes or delegates the task further to specific handlers. AppCore is also responsible for initiating parts of the OpenGL rendering, shown at the bottom of Figure 12.
The last primary class is PlotManager, but given that this is a static class there is no creation. Its initiation is instead done by a function invocation, `PlotManager::Init()` (shown in Figure 13), which is performed by AppCore when a file is loaded, or when several different commands are registered. Despite the function’s name, it is not used just once but it is instead used primarily to force a new OpenGL render and to update some states.

4.1.2 The main rendering loop

As presented in the previous section, autoUpdateGL() is the main rendering loop (shown in Figure 11). The conditional of the first opt-fragment, `IsUpdateGL or IsAnimating`, is a state check which is true if `Intplot3` has scheduled a repaint, or turned on animations, as a result of a registered command. If the conditional is true and if GLWidget is not currently being painted on, then updateGL() will forward the application’s intent to trigger a repaint of GLWidget to Qt. This will eventually lead to GLWidget’s `paintGL()` function to be invoked by the Qt repaint-event; part of the function is presented in Figure 14. The full diagram can be found in Appendix D.
The purpose of the paintGL() function is to clear the previous render, setup the OpenGL view and matrices and initiate the new render call hierarchy. As seen in Figure 14, the outer alt-fragment is what decides on whether or not the render will be performed by the old or new core, based on set states. These states are set during runtime and they are tied to what commands are inputted by the user, and after being set they are picked up by the repeating render loop.

Focusing on the new core rendering in Figure 14, if the states are set in such a way then the else-conditional of the outer alt-fragment will invoke GLWidget’s renderOnCurrentContext() function. The function is shown in Figure 15 and it is the last high-level call before having
PlotManager iterate over the Intsyspp models, which initiates the final render call hierarchy for each element that needs to be rendered.

![Diagram of the function GLWidget::renderOnCurrentContext()](image)

In Figure 15 we see in the alt-fragment that PlotManager’s function \textit{Plot()} is called. This function, detailed in Figure 16, have two sets of seemingly different plot calls, shown here as \textit{PlotFaceOGL()} in the beginning and then \textit{PlotFaceOGL2D} at the end. The difference between the OGL and OGL2D calls are that some element types in Intplot3 has no three-dimensional representation, for example a circle. These elements and three-dimensional ones are rendered in the functions denoted 2D and without the 2D suffix respectively.

It is important to note that the first alt-fragment is one of several. GPlot (Geometry Plot) is the default rendering command of the new core, which is why the other fragments were left out, to reduce bloat in the diagram.
Figure 16. Diagram of the function PlotManager::Plot()

Given the scenario of GPlot being the active rendering command and that raster graphics are enabled, then the function *PlotFaceOGL()* will be invoked, see Figure 17.
There exists four different custom data structures used to manage the Intsyspp models, they are (in the order of highest to lowest level): Assembly, FileSet, Model and Element. Each level contains a list which holds objects of the lower level data structure, Assembly for example has a list of FileSets, each FileSet has a list of Models, and so on. The informal data model category mentioned earlier (Figure 6) refers to these four classes, where Assembly is the class that encapsulates the entire model. PlotFaceOGL() iterates over all FileSets, all Models and all Elements as shown in Figure 17. All elements that are selected are to be rendered using OpenGL, which is performed in the function PlotFace() shown in Figure 18. It is a “loose” function, which means that it is not part of a class.

PlotFace() is almost at the end of the render call hierarchy. The element to be rendered is sent to the PlotFace() function and using a series of if-else statements the element is finally passed, in this example, to PlotFace_Volume3D(). It is at this stage that the final OpenGL calls are executed and the element is rendered. This final process is shown in Figure 18.
4.2 Identified needs and insufficiencies

Based on the code review, the generated system documentation and discussions with the supervisor at Saab three main needs were identified for Intplot3, given its current state, that were applicable to the focus of this work. The needs were then associated to two software quality attributes: (1) Maintainability and (2) Performance efficiency. The three identified primary needs are:

- The rendering structure need to be made clearer and less distributed.
- OpenGL needs to be upgradeable.
- The rendering performance needs to scale better for large finite element models.

These three needs were selected as they represent the most extensive issues with the code base today. In addition to the list of needs there are insufficiencies within the current design, there are violations to the SOLID design principles which makes the code more difficult to understand. These insufficiencies should preferably be addressed in the proposed system design, for example, there exists God classes that violates the Single responsibility principle.

4.3 An improved system design for Intplot3

Based on the class and sequence diagrams of the system documentation it is clear that the structure is in need of clarification and refactoring in order to meet the identified needs. For instance, AppCore and GLWidget are anti-patterns of the type God class and they violate the Single responsibility principle of SOLID. Much rendering functionality is spread throughout Intplot3 as discerned from Figure 6 and the generated sequence diagrams. Additionally, this widespread rendering functionality is often tightly coupled with legacy OpenGL, making upgrades difficult. Looking at Figure 17 and 18 we note a potential performance bottleneck: many rendering calls are unnecessary, or undone, based on what the following element type is. For example, disabling lighting in Figure 18 is unnecessary if the next element’s type supports lighting, but since elements are currently ordered by their id there is no way of optimizing this
process cleanly as is. For a small size model this is no discernable performance loss, but if a model in total contains potentially one or two hundred thousand elements (which is possible), then these types of calls could become a noticeable issue.

In order to make the structure less complex and less distributed, to meet identified needs and address insufficiencies, the structure presented in Figure 19 is proposed. The new high-level structure aims to improve the existing component-based architectural style, and as such it introduces three new components: Canvas-, Core- and RenderComponent. The CanvasComponent is what is drawn on and interacted with, and the CoreComponent is responsible for application logic. Lastly, the RenderComponent (and the focus of the proposal) is responsible for the custom rendering logic in Intplot3. All class diagrams are only concerned with the ownership-relations between the classes and components, as the interactions are too complex for the scope of the proposal to be modeled.

![Figure 19. High-level diagram of the proposed system design](image)

The internal structure of the CanvasComponent is illustrated in Figure 20. Compared to the existing system class diagram (Figure 7) there is no visual difference, all relations are the same. We see that MainWindow owns the CanvasComponent in Figure 19, which translates to it owning the DockableWidget in practice, same as in Figure 7. The reason for it being proposed as a component is due to GLWidget’s underlying characteristic: to be interchangeable.

![Figure 20. Class diagram of the CanvasComponent](image)

Next is the CoreComponent, illustrated in Figure 21. Comparing the new structure to the existing one (Figure 7) we see that AppCore has been split into three separate classes: AppCore, CmdCallback and OldCore, all owned by IntplotCoreDelegate. The ownership relation of the CoreComponent shown in Figure 19 is in practice between MainWindow and IntplotCoreDelegate. All calls between the three new classes should go through IntplotCoreDelegate in order to reduce coupling to exterior classes. The new OldCore is meant to encapsulate all interactions with old core functionality in Intplot3. Depending on the size of this task OldCore may in fact need to be regarded as an entire component, instead of a single class. It can in that case be implemented the same way as the new RenderComponent.
Lastly, the third component is the RenderComponent, presented in Figure 22. Being the focus of the proposal, its purpose is to encapsulate all custom render logic inside Intplot3, including all OpenGL calls. The interior contents of the component shown in Figure 22 represents classes or lists of classes. Design-wise there are two classes accessible from outside the component structure, Render and RenderState. The ownership-relation between MainWindow and RenderComponent seen in Figure 19 is, at a lower level, between MainWindow and the new class Render.

Render is based on the Facade pattern and it is what encapsulates all render functionality as seen from outside the component. RenderState will be a static class encompassing all render-related states that currently exists in Intplot3. As such, the main workflow of Intplot3 will remain unchanged using this design, the rendering will still be based on the usage of states which can be accessed the same way they are today.
The interior structure of the new RenderComponent is separated into three core sets of classes: `Logic`, `Helpers` and `OpenGL`. All names shown in Figure 22 are examples of class names that represent functionality that currently exists, and where they could be placed. The class `RenderManager` is almost the equivalent of `PlotManager` today, it is the highest level which will delegate the rendering call to sub-managers, each with their own responsibilities. For instance, `PlotManager` (new class) in Logic is only concerned about drawing the three-dimensional model, the MarginManager is only concerned with drawing the margin, and so on. Logic-classes are high-level who relies on medium level Helper-classes, or directly on low-level OpenGL classes, to perform their tasks.

The `Primitive` class in OpenGL contains core draw-functions such as `DrawTriangle()`, `DrawCurve()` and `DrawCircle()`. `DynamicPlot` exists for the reason of allowing objects such as texts to be added or removed dynamically to a list whose contents is then rendered.

Absent from Figure 22 is any mention of the old core, despite there being two modes of rendering in Intplot3. The old core contains all its own rendering functionality, in order to initiate a render using the old code `RenderManager` can divert the render call to `OldCore`, which in principle is how it works today.

Due to the fact that Qt issues events such as paint and resize to `GLWidget`, for the RenderComponent to work these events must subsequently trigger appropriate handlers inside the class `Render`. For example, `GLWidget::paintGL()` should invoke a similar `Render::Paint()` function, which in turn invoke a similar function in `RenderManager`. This is basically the same as the existing `renderOnCurrentContext()` invoking `PlotManager::Plot()` as seen in Figure 15. In order to reduce coupling, `MainWindow` will issue the necessary connects similar to what already happens (see connects in Figure 9).

The reason for all these changes is to enforce the Single responsibility principle which is necessary to reduce code complexity inside Intplot3, as currently several classes do not adhere to said principle. Also, the changes makes the structure of Intplot3 clearer, not as distributed and more logical, as functionality is located based on classes’ responsibilities.

Separating the current `AppCore` into the three new classes makes them able to adhere to the Single responsibility principle, and `AppCore` will no longer be a God class, or at least to a much lesser extent. Despite `GLWidget` being identified as a God class as well it does not need to be split into separate classes. Its wide range of responsibilities will instead be refactored into appropriate classes in the new `RenderComponent`. What is left in `GLWidget` after this process will essentially only be interaction logic.

Detailing the proposed system design further, Figure 23 illustrates a (very) limited pseudo-code example of the `Paint()` function inside `RenderManager`, which is ultimately invoked every time Qt issues a repaint-event to `GLWidget`. The normal rendering process will only rely on Qt’s repaint-events, the existing `autoUpdateGL()` loop will only be used during animations. Said loop will be moved to `RenderManager` and it will start or stop depending on the `IsAnimating` state. The loop will indirectly invoke the `Paint()` function as normal, according to Qt recommendations, using `updateGL()` as shown in Figure 11.
Using the approach outlined in Figure 23, all custom-rendered GUI components, such as the three-dimensional model and the margin, are rendered in a clear and structured way based on a number of set states. The proposed design is based on the fact that the element list to be rendered is ordered by element type. This reduces unnecessary OpenGL calls and state changes due to grouping, thus improving performance. Currently, as shown at the top in Figure 18, each element generates a function call. The pseudo-code example passes the list of elements as a pointer, with start and stop ids, as to minimize the number of excessive calls. It can also help with grouping commands in the relevant Helper- and OpenGL-classes.
5 Discussion

This thesis has presented a system documentation of Intplot3 and a proposal for an improved system architecture and design based on identified needs and insufficiencies. The following chapter will discuss the generated results and analyze the proposed system design.

5.1 Results

As mentioned initially, the size of Intplot3 and the number of complicated relations to linked libraries entailed that, in relation to the time limit of the thesis, the results had to be of limited granularity. The documentation provides an overview of a select set of classes in Intplot3, deemed important for understanding the main interactions. In the presented class and sequence diagrams several simplifications were made as to be able to provide a relevant system overview, without losing too much detail. Often similar calls were grouped under a descriptive name. Function arguments were left out whenever not necessary. All diagrams were focused on the custom rendering functionality in Intplot3.

The proposal for the improved system design was based on the generated documentation and was as such also primarily focused on the custom rendering. The basis for all design decisions made during development were always focused on the set of identified needs and insufficiencies, as well as the associated software quality attributes.

5.1.1 Evaluation of the improved system design

In order to evaluate the proposal; a qualitative approach will be used since the design is too high-level for design metrics to be applied according to the discussions in 2.4. Recapturing the results of the identification process, there were three identified primary needs:

- The rendering structure need to be made clearer and less distributed.
- OpenGL needs to be upgradeable.
- The rendering performance needs to scale better for large finite element models.

These three needs were associated to two software quality attributes: Maintainability and Performance efficiency. There was also insufficiencies with the current system design in the form of identified SOLID design violations. The needs, insufficiencies and software quality attributes will be discussed in relation to the proposal presented in Figure 19 to 22.

In the proposal there were three new components that were introduced, Canvas-, Core- and RenderComponent. While the Canvas- and CoreComponent relates to the first two listed needs, as their contained classes today contain render-functionality, they are primarily designed to address the identified insufficiencies. The RenderComponent is in practice what allows the new design to meet the three needs.

In order to meet the first two primary needs the design refactors all custom rendering-related functionality into a dedicated component, the RenderComponent, inside of which all OpenGL calls are performed in a limited set of classes. This makes the OpenGL logic easier to be upgraded or replaced in the future. The separation of concerns helps clarify the structure as all classes have defined responsibilities. Refactoring the render functionality also makes it less distributed, as it is contained in fewer but more cohesive classes. The new logically separated structure shown in Figure 22 should make interactions easier to understand and follow. Based on these design decisions the first two primary needs are deemed to be met.

The decisions made in order to meet the third primary need were based on potential performance bottlenecks identified during the manual code review, as noted in section 4.3. The design decisions were made to reduce the amount of OpenGL calls overall, especially to reduce
the number of unnecessary OpenGL state-changes. These changes are made possible by ordering elements by their types (and using references to the lists). The changes could provide a noticeable difference in rendering speed for large models, based on the theory presented in section 2.3, but without measurements this is unsubstantiated. However, given the sheer number of reduced OpenGL calls, as an effect of this design, it is reasonable to think that there would be a performance boost. Based on this sentiment, the third need is deemed to be met as the new design is estimated to result in a performance boost.

Lastly we have the identified insufficiencies in the form of SOLID violations, designed to be addressed by the proposed Canvas- and CoreComponent. As mentioned initially in 4.3, the existing AppCore and GLWidget classes are anti-patterns in the form of God classes that violate the Single responsibility principle. The proposed high-level solution to these insufficiencies is to implement the two classes as components, as illustrated in Figure 20 and 21.

The AppCore that exists in the current design is to be split into three classes, AppCore, CmdCallback and OldCore (the latter one may potentially be regarded as a component). This makes for a much clearer separation of concerns which should make each of them adhere to the Single responsibility principle, at least to a larger extent than currently.

As illustrated in Figure 20, and mentioned in section 4.3, GLWidget does not require a split in order to adhere to the Single responsibility principle, even though it currently is a God class. The reason for this is that most of the wide range of functionality in GLWidget is either concerned with interaction or rendering. Given the fact that all custom rendering is refactored into the new RenderComponent, using this design, means that essentially only interaction logic will remain in GLWidget. Introducing GLWidget and its related classes, shown in Figure 20, as a component will however make it easier to adhere to the Single responsibility principle in future development.

While the suggestions for improvements regarding the Canvas- and CoreComponent are abstract, the affected classes should at a high level no longer be violating the Single responsibility principle, which makes them adhere better to the SOLID principles. This ambiguity entails that the classes at a deeper level may still violate some principle and that there may still be improvements to be made. Without a thorough code review however, which is not within the focus of the proposal, the identified insufficiencies are (at a principle level) deemed to be met.

The improved system design proposal is as mentioned based on identified needs, but it is also at first hand based on the existing design, meaning that all changes need to be viable. As much code as possible should be usable in the new design to minimize the costs of implementation. Based on the performed manual code review, most classes and their respective responsibilities and functionality shown in Figure 22 already exist in Intplot3. Implementation is mostly a matter of relocating, renaming and/or refactoring them. The intent of Figure 22 is primarily to show the intended component structure.

The first two primary needs and the insufficiencies are related to the quality attribute Maintainability, the third need is related to Performance efficiency, which has already been discussed. The proposed system design should increase class cohesion and encapsulation, while simultaneously reducing coupling and complexity. These four affected design properties should, according to QMOOD increase code Reusability, Flexibility and Understandability. Based on the similarities between the quality attribute definitions of QMOOD and the ISO/IEC 25010, discussed in section 2.4, it stands to reason that the Maintainability of the system should benefit from the new system design.

Based on these conclusions it is reasonable to expect that the proposed system design should improve the Intplot3 application and make it more easily maintained. The new system design is also expected provide a performance boost, especially for large finite element models.
5.2 Method

Most of the work presented in this thesis is the result of a manual approach. The code was analyzed and reviewed manually; the diagrams were also generated manually. This allowed for a high degree of validity and reliability since the work was not general in nature, as would be the case (to some extent) if automated tools were used. The manual work was very purpose-driven, and relevancy was decided by a person based on previous findings as well as the aim of the thesis. However, not using automated tools substantially limited the scope of the work, as the manual approach required a lot of time.

As mentioned in 3.1, there was a lack of automated tools available. This was not a conscious choice. The tool Enterprise Architect was used but it offered limited (relevant) functionality. It was as such only used to generate an outline that could guide the manual code review. The work was performed at Saab, which meant that it was done in Saab’s secure computer environment. This means that all software must go through an approval process, which can take a long time. In the case of this thesis: no new tools could be approved in time, and the number of tools that were available was limited. Enterprise Architect was the most suitable tool available.

Given the manual approach: time was the main limiter of the scope of the work. The aim of the thesis was to document a large part of the Intplot3 application, this meant that the depth had to be limited due to lack of time. Consequently, the documentation and the generated proposal are both on a relatively high level, with some generalizations having been made.

If automated tools had been used to generate diagrams and analyze the code, the time could instead have been used to increase the scope of the work. Increasing the scope could also benefit the proposed design, as the suggestions would be based on a more complete documentation.

5.3 Future work

Currently many global states exist in Intplot3 which is typically regarded as bad practice in an object-oriented application, although sometimes necessary. It is recommended that they are reduced in numbers in order to reduce code complexity, and that those that remain are documented. Many states are needed today because commands are not persistent objects, and neither are their dialog counterparts. These states may be refactored into data objects which a command or its dialog alters. Utilizing the Observer pattern (based on Qt connects), the solution may even allow for synchronization between command input and the GUI dialog, which today is a complex issue.

Version 4.8 of the Qt framework is, as mentioned earlier, used in Intplot3. This version has been unsupported since the end of 2015 [26]. It is therefore recommended that Qt is upgraded to a more recent version.

The proposed design is focused on improving the existing system, so while it only attempts to improve on the usage of fixed function pipeline functionality it is strongly recommended that Shaders are introduced in the future. Not only will it give a greater flexibility to rendering, it will offload much of the CPU usage to the GPU. The current CPU utilization seems to be the main performance bottleneck when rendering large models, since Intplot3 is not threaded. The introduction of Shaders may alter the proposed design, but the key principles should still be mainly applicable, and the overall component structure should still be valid.

5.4 The work in a wider context

The need today for maintainable large-scale systems is evident. Not only can system complexity increase development costs, it can also introduce errors. Intplot3 is used for visualizing results from calculated stress tests that has been and will be used for many years. This motivates the need for making the code base maintainable, that is easy to analyze, modify and that can perform well without error. Since the visualization is based on calculated results it is important that the
application has great control over what, where and how something is rendered, as the results are used by stress engineers. Using the proposed system design, Intplot3 can be made more maintainable and the new render component can give the developers a high degree of control over the custom rendering.
6 Conclusion

The aim of this thesis was to create a model of the existing system architecture and design of the application Intplot3, and then generate suggestions for improvements. These tasks were formulated as three research questions:

- What does the existing system architecture and design look like at a high level?
- What needs and insufficiencies exist for, and in, the existing system?
- How can the system architecture and design be improved?

In order to answer the research questions a manual code review approach was primarily used, and the proposal was generated iteratively in collaboration with the supervisor at Saab. This approach ensured a documentation that was representative of the current system design, and a proposal that could be viably implemented.

The improved system design was evaluated using the set of high-level system needs, associated quality attributes and identified insufficiencies that acted as the basis for the new design. The results of the evaluation concluded that the proposal met the identified needs, and that the insufficiencies had been improved upon, at a high level. The proposed changes were deemed to be able to increase both the maintainability of the system and the performance efficiency of the custom rendering in Intplot3. The proposal improved on the existing system architecture by introducing components.

Given the scope of the work and the size of the system, both the documented model and the proposal are on a high level, with some parts based on simplifications or generalizations. While necessary in order to be able to capture all the important aspects, it naturally leads to some ambiguity. Yet, the model do provide an accurate overview of the main parts of Intplot3 and can effectively be used to understand large parts of the existing system. The proposal was generated carefully as to be able to meet all high-level needs and address identified insufficiencies in the system, as concluded; it can therefore be used to guide the future development process.
Bibliography


Appendix B - GLWidget

Appendix C - IntplotCoreDelegate
Appendix D - GLWidget::paintGL()