TAMPEREEN TEKNILLINEN YLIOPISTO

ARI AALTONEN

PRODUCT-LINE ARCHITECTURE FOR A VISUALIZATION SYSTEM

Thesis

Tarkastajat: professori Kai Koskimies
professori Jouni Mattila

Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekuntaneuvoston
kokouksessa 6. huhtikuuta 2011

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
AALTONEN, ARI: Tuoterunkoarkkitehtuuri visualisointi järjestelmälle
Diplomityö, 47 sivua, 4 liitesivua
Huhtikuu 2014
Pääaine: Ohjelmistotuotanto
Tarkastajat: professori Kai Koskimies, professori Jouni Mattila
Avainsanat: Ohjelmistoarkkitehtuuri, tuoterunko, Qt

Ohjelmistoarkkitehtuuri on tärkeä osa kaikkien suurten ja monimutkaisten ohjelmistojen projekti ohjelmiston kehitysprosessia. Joissakin tapauksissa, samaa ohjelmistoarkkitehtuuria käytetään toteuttamaan useampia ohjelmistotuotteita käyttäen niiden yhteisiä komponentteja. Tämä mahdollistaa ja tarvitsee tuoterunkoarkkitehtuurin (PLA) suunnittelun, jotta kokonaisuus pysyy hallinnassa.

PLA on tuotealusta, joka sisältää monien samaan toimialueeseen kuuluvien ohjelmistotuotteiden yhtäläiset piirteet. Käytettäessä samoja komponentteja uudelleen eri tuotteissa ne ovat vakaampia johtuen kattavammasta testauksesta, ja tuotteen markkinoille valmistuminen nopeutuu. Se tarjoaa variaatiopisteitä, joissa uudet ominaisuudet voidaan turvallisesti toteuttaa.

Tämä väitöskirja perustuu suunniteltuun 3D-visualisointi ohjelmistoon, joka kasvoi tuoterungoksi toteutettaessa muutamia sovelluksia, jotka hyödynsivät yhteisiä komponentteja ja arkkitehtuuria. Kyseinen tuoterunkoarkkitehtuuri on plug-in-malliin perustuva järjestelmä, joka rakentuu Malli/Näkymä- ja viestinvälityssuunnittelumallin varaan. Nämä kolme arkkitehtuurista mallia muodostavat yhdessä modulaarisen, muokattavan ja laajennettavan tuoterunkoarkkitehtuurin.

Tässä työssä kuvataan arkkitehtuuri sekä arkkitehtuuriset variaatiopisteet. Tässä arkkitehtuurissa on kahdenlaisia variaatiopisteitä, sisäisiä- ja päävariaatiopisteitä. Sisäiset variaatiopisteet on tarkoitettu kehittäjille, jotka integroivat uusia ominaisuuksia tuoterunkoon ta luoda uusia sovelluksia. Päävariaatiopisteiden tarkoituksena on laajentaa sovelluksia, mutta niitä voidaan käyttää myös lisäämään kokonaan uusia ominaisuuksia sovellukseen. Tuoterunkoarkkitehtuuri tarjoaa myös APIn ja kirjastoja mahdollistamaan ja helpottamaan uusien laajennuksien toteuttamista.

Tällä tuoterunkoarkkitehtuurilla toteutetut sovellukset toimivat tässä tapauksessa testitapauksina ja niitä käytetään arvioimaan tuoterunkoarkkitehtuuri. Sovellusten testaus on osoittanut niiden täyttävän niiltä vaaditut laatuvaatimukset. Nämä testitapaukset näyttävät tuoterunkoarkkitehtuurin olevan riittävän muunneltava, laajennettava ja tehokas mahdollistaen sovelluksien toteuttamisen ja niiden vaatimuksien täyttämisen.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
AALTONEN, ARI: Product-line architecture for a visualization system
Master of Science Thesis, 47 pages, 4 Appendix pages
April 2014
Major: Software engineering
Examiner: professori Kai Koskimies, professori Jouni Mattila
Keywords: software architecture, product line, Qt

Software architecture is an important part of the software development process of any large and complex software project. In some cases, the same architecture is used to implement more than one software products using many common components. This enables and requires the designing of product line architecture (PLA) to keep it all under control.

PLA provides variation points where new and different features can be safely implemented. It is a product platform that contains the commonalities of many software products of the same domain. This way the implementations are more robust as they are tested more thoroughly and time-to-market is decreased.

This thesis is based on the design of 3D visualization software that grew to a product line for implementing few applications on the common components and architecture. The PLA is plug-in based system that has two underlying patterns it is built on, model/view and message dispatcher. These three architectural patterns are combined to form a modular, customizable and extensible PLA.

This thesis describes the architecture as well as the architectural variation points. There are two kinds of variation points, internal and main. The internal variation points are meant for the developers integrating new features into the product-line or develop new software products. The main variation points are aimed at extending the applications, but may also be used to add completely new features to the application. The PLA also provides APIs and libraries to enable and make it easier to develop new plug-ins.

In this case, the applications designed using this PLA work as test cases and they are used to evaluate the PLA. Testing the applications has shown they fulfil the quality requirements they were given. These test cases show PLA to be modifiable, extensible and efficient enough making it possible to build the applications and meet their requirements.

FOREWORDS

I want to thank my colleagues for their support and the examiners of this thesis Kai Koskimies and Jouni Mattila for their guidance for this work. I also want to thank my family and friends for their question about the status of this thesis.

Tampere, March 25, 2014

_____

Ari Aaltonen

TABLE OF CONTENTS

ACRONYMS & DEFINITIONS

API	Application Programming Interface
DDS	Data Distribution Service for real-time systems
DLL	Dynamic Link Library, library file that is loaded as a part of a program during execution.
GUI	Graphical User Interface
GUID	Globally Unique Identifier
IHA3D	Visualization software developed at Department of Intelligent Hydraulics and Automation at Tampere University of Technology.
MFC	Microsoft Foundation Class Library is a library that encapsulates part of Windows API interfaces in to C++ classes.
Model/View	Architectural design pattern similar to MVC used in Qt
MVC	Model-View-Controller
Ogre	Object-oriented Graphics Rendering Engine is used to render 3D graphics on a window.
Plug-in	Dynamically loadable program component that is used to extend existing software or provide a way of implementing variability
Qt	cross-platform application development framework
SDK	Software Development Kit
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

1. INTRODUCTION

The purpose of this work is to introduce the design of IHA3D PLA and evaluate its architecture through the three applications developed using it. It is related to International Thermonuclear Experimental Reactor (ITER) project Intelligent Hydraulics and Automation (IHA) department of Tampere University of Technology (TUT) is involved with.

ITER is a large-scale multinational experiment aiming to prove that fusion is commercially viable option as an energy source. The aim is to produce 10 times more energy than keeping up the reaction consumes and to keep the reaction going for 1000 seconds. It's used for testing materials, developing technology and collecting data for building the first electricity-producing fusion power plant. The preliminary testing of a maintenance robot for the reactor is done in Tampere. One of the software requirements for the project was to develop 3D visualization software.

Data visualization is part of many modern software products. In this case it is used in the visualization of a robot to view a 3D representation of the real world environment. The virtual environment can be displayed from different angles and user can move around in it to get a desired view of the environment. The views can display additional variables of the state of the robot presenting important information to help in decision making situations. In some cases there is camera input presenting a real-time video feed of the robot's environment. This can be displayed to the user as it is. The video can also be augmented with visualizations of the data the system produces while in operation.

Virtual environment provides the user a way to practise the robot operation tasks by showing effects of the robot control input in the virtual model. This way he can practice the control operations off-line, familiarizing him on the controlling tasks beforehand. This helps to prevent human errors on the job. Also it is not always possible for an operator to be present while using robotic systems because of possibly hazardous environment. For this reason, it is important to have a visualization system presenting the robot in the operating environment.

Visualization system may have different requirements depending on the purpose it is used for. It could be used to construct the 3D environment and the robot, or just to operate the robot in its environment. These two examples have some common features and many specialized ones. Also projects can grow and change with time if it's large and divided into phases, each phase having some new feature requirements. Thus it is useful to have a flexible architecture for the system which can be extended and changed using different plug-ins. The main advantage of using a plug-in system is the improvement in modularity.

Software architecture design is an important part of software development process. Software architecture is a top-level description of the structure of the software. It deals with high-level decisions to maintain system integrity keeping the software in a unified form. Its purpose is to constrain and guide the development of different software components. It identifies the software's structural components and defines the interfaces they offer. The interfaces define the communication between components and specify their externally visible properties and behavior. Architecture addresses crosscutting concerns that cannot be made by a developer with a narrow focus of responsibility such as designing one software component.

Product-line architecture (PLA) is a product platform built with API interfaces and component libraries. It contains the commonalities of many software products of the same domain and provides points of planned variability. It has many good properties, but also some downsides. ([1], p. 157-186)

One of the good properties of PLA is that it can develop higher quality of products as the same tested code components are reused. The time-to-market is also faster as less code is needed to implement a new product. Standardization comes as a by-product as products work the same way. Using PLA lowers labor needs as the architectural design has already been done, at least the major part of it. From the project management point-of-view, PLA makes easier project management for single products as same kind of development process is used in other projects. It also lessens the need for staff re-education when moving them between projects as the tools and environment stay the same. ([1], p. 157-186)

The problem with product-line architecture is that it takes a lot of time and resources to design and implement. Many products need to be implemented using it before the effort starts paying back. ([1], p. 157-186)

IHA3D PLA was developed from the common components of applications using the same architecture. There are three applications of IHA3D at the moment.

The first is the reason of the development. It was developed for visualizing the maintenance robot’s movement and surroundings in the reactor. It was designed to work as a part of a distributed system controlling and monitoring a service robot in radioactive fusion reactor conditions. This configuration is meant to be used on a multi-core Windows computer with high-speed Ethernet connection.

The other application is for educational purposes. It is limited in functionality but has other redeeming features that fit for gaming and presentation. It was designed to work on a slow single-core laptop and though it can utilize the internet for content, it is not required. This configuration works even on a slow single-core Windows computer with no need for network access.

The third application is a very basic setting that has almost everything stripped down or automated for easy use. It is used in a course project for teaching simulation of

hydraulics. The application is used to visualize the movement of a boom in response to the student's control software running on a separate real-time simulation environment. This application was further evolved for Comau industrial robot visualization. These configurations don't require as much processing power as the first application but use the Ethernet connection for control commands.

Chapter 2 gives background information on Qt SDK which is used for implementing the applications. The chapter 3 introduces the existing system and describes the quality requirements. Chapter 4 describes the architectural design of the software and explains the most important commonly used interfaces in the API and the software modules. Chapter 5 describes the implemented applications. Chapter 6 gives the results of the applications in action. Chapter 7 describes how well the PLA meets the requirements. Chapter 8 contains the conclusions.

2. OVERVIEW OF QT

This chapter gives a brief overview of Qt SDK and the way things are done using it. Qt is a full cross-platform development framework providing tools and libraries for creating applications and user interfaces. It is well documented, has an active community and is constantly developed. These qualities make Qt an excellent choice for developing many kinds of applications. [16]

When using an SDK in the development, it is best to follow its coding conventions and use the patterns it provides. Qt provides too many things to list here, so this chapter will only include the parts deemed most usable to this work.

First subchapter discusses the use of implicit sharing [2]. Then the concept of signals and slots is described [3]. Next, the concept of Event Driven architecture is visited briefly [8] before explaining the idea of event processing [4]. After that we move on to patterns in general [5]. The first and one of the most commonly used patterns is the Observer pattern [6]. After that the Dispatcher pattern is discussed [17]. The Pimpl idiom is explained thoroughly in subchapter 2.4.3 [7] [15]. Next the concept of Event Driven architecture is visited briefly [8]. At the end the Model-View-Controller pattern and Model/View pattern are described [9] ([1], p. 142).

2.1. Implicit sharing

Qt uses implicit sharing, atomic reference counting and reentrancy for many of its value classes to maximize resource usage and minimize copying and CPU usage combined with ease of usage.

Implicit sharing is another name for a more common term copy-on-write. The copies all look like different variables, but in reality, they all point to the same data. Only when user changes its copy of the data, the modified data is copied to a different place. This delays the possibly CPU-intensive copying, and in most cases, removes need for it completely. All this is hidden from the user of the data structure by encapsulating it behind an interface. Instances of the data structure presented by the interface can be used as normal.

Qt uses atomic reference counting with implicit sharing. Atomic reference counting is like normal reference counting. It only adds the guarantee that the counting variable updates are serialized. They are used to keep count of the references to the implicitly shared data. This makes it possible to use implicit sharing with multiple threads, though it does not ensure thread-safety on its own.

Qt makes extensive use of reentrant functions. Reentrancy basically means that the member functions of a class only access the member data of the class. Reentrant classes are safe to use across threads as long as the function calls are serialized.

2.2. Signals and Slots

Signals and slots are a Qt way of implementing communication between objects. It is used to pass data and make function calls by implementing the observer pattern. This is a loosely coupled solution where the signal sender knows nothing about the receiving object. This is more flexible than generally used function pointers. Function pointers are not type-safe as there is no way to be sure the parameters are correct. Function pointers also force strong coupling because the processing object must know what type of callback function to call.

Qt signals and slots usefulness extends to the multithreaded systems. They behave as direct function calls if the signal sender and the recipient are in the same thread. If the sender and recipient are in different threads, the function call (signal) is queued and executed when the recipient thread has time to process it. Signals and slots are the recommended way to pass data between different threads as it can be done without locking the data explicitly.

Meta object compiler (MOC) is used in Qt to generate the necessary boilerplate code for the signals and slots defined in classes. This makes signals and slots easy to use. Qt widget and object base classes have many predefined signals and slots, but it is common to subclass them and add new behavior.

2.3. Event-processing

The main ideas behind Event driven architecture are loose coupling and distribution. It provides an excellent basis for scalability. It is based on message dispatcher pattern and requires only implementation changes to make it threaded and thus improving for example GUI responsiveness.

Qt events are usually a result of a GUI action when the OS sends the application a message which is translated into Qt event object for handling. Another source of events is the application itself. Events can be generated for example using timers to execute actions periodically.

Every thread in Qt that handles events or queued signals runs its own event loop. This enables easier multithreaded application development as developers don’t need to worry so much about locking. Any object inheriting from QObject base class can receive and handle events. The handling is done by re-implementing the needed handler functions in the subclass. Some events, like key events, can be propagated and are first sent to the GUI element most likely to handle it. The event is propagated on to a next element if the event is not handled by that object.

Qt event processing system can be extended with custom events. The new custom events inherit an abstract QCustomEvent class and implement their own interfaces

which are accessible to the receivers after casting to a right type. These are handled by the receiving object in its custom event handling function.

2.4. Patterns

Software design pattern is defined as general repeatable solution to a commonly-occurring problem. They are used to help with coupling and cohesion issues, improve reuse of code and maintainability and many other problems. Design patterns can be used on many levels of software design. Higher level pattern are called architectural patterns. They are larger scope patterns that usually describe the overall structure of an entire system. The lower level patterns are just called design patterns and they are meant to solve some local problem concerning a software component or some part of it.

2.4.1. Observer pattern

Observer pattern defines one-to-many dependency between objects allowing a Subject object to notify registered Observer objects of changes in its state. Observer patterns can be implemented in two ways, push or pull types. In pull type implementation, the Subject only notifies the observing objects of a change, which in turn get the new data from the Subject if necessary. In push type implementation, the Subject sends the changed data to the observing objects. The push type implementation compromises reuse as the same code has to be modified to send other data in different places. The pull implementation on the other hand is less efficient as it requires observers to call the Subjects getter-functions to get the changed data in addition to the added delay in informing observers of the change.

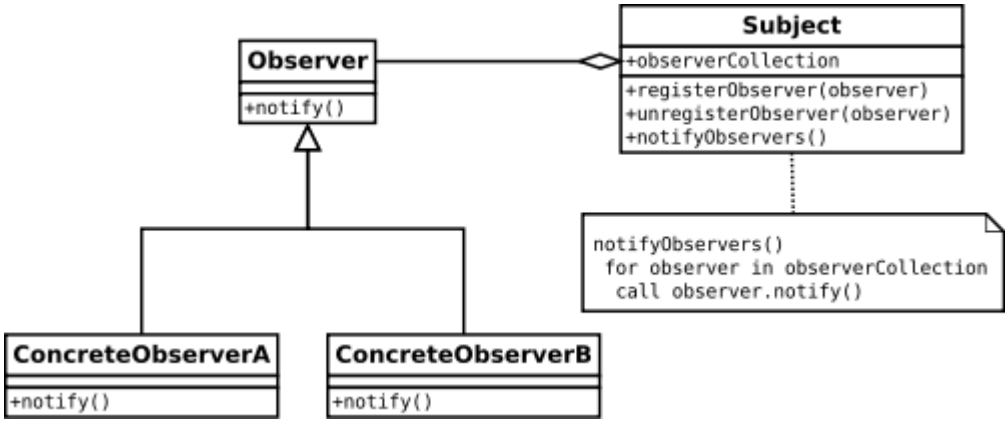

Figure 2.1. Observer pattern [6]

This pattern is in central role in Dispatcher pattern and Event-processing architecture, MVC and Model/View architectures.

2.4.2. Dispatcher pattern

The Dispatcher is a convenient pattern when the component interfaces are not all known beforehand. The component specialties don't have to be known. This gives more freedom for component designers.

The data structures that are transmitted between the component and the Dispatcher don't have to be defined. Communicating components all have a common interface they implement. This interface is registered to the Dispatcher module using the interface it provides. In some cases, the components can register to receive only certain kinds of messages concerning their interests.

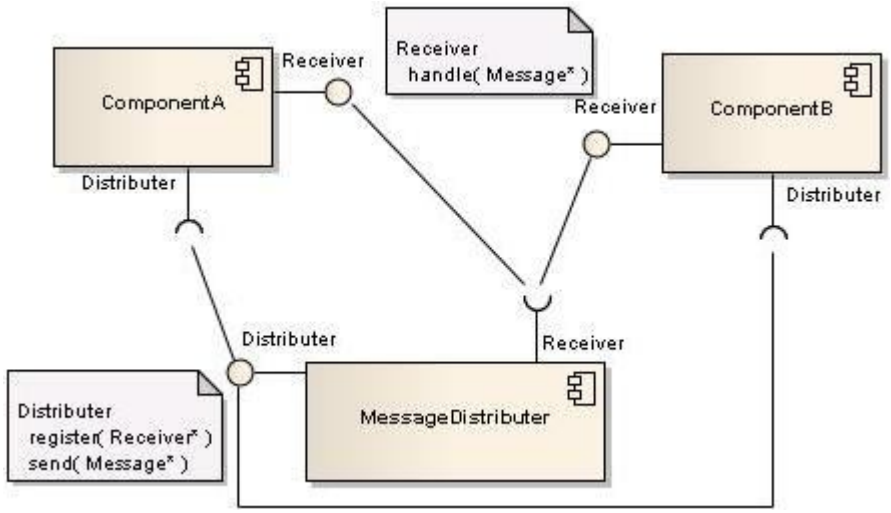

Figure 2.2. Dispatcher pattern

The message handling interfaces don't have to be changed if a developer defines a new kind of message. The only true requirements are that the new messages follow the interface and structure defined for them and they can be distinguished from the other messages. The Dispatcher and the components can work in parallel though it naturally requires the component developer to take this in to account.

This pattern is central to the Event-processing architecture.

2.4.3. Pimpl idiom

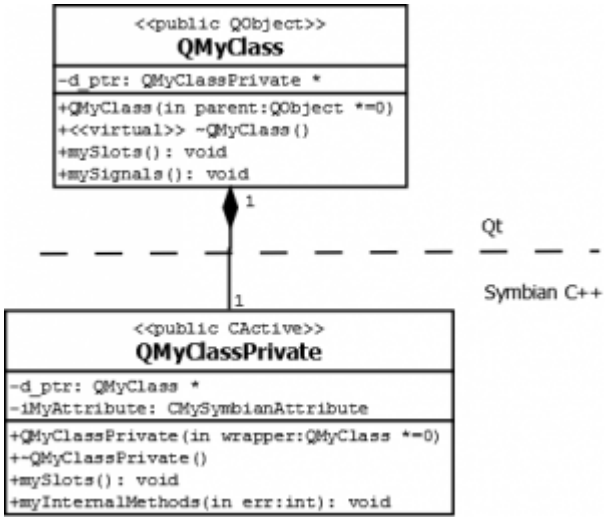

Figure 2.3. Pimpl idiom

Pimpl idiom is sometimes called opaque pointer or handle classes. Its main purpose is to provide source code compatibility hiding all the operating system specific functionality. It is used to hide data structures and class implementations from the users of the class interface. Other reason is to reduce dependencies between classes thus reducing compile time of the software. It also allows the implementation to be changed without the need to recompile every component using it. The implementation part could also be changed during the program's execution. Qt uses Pimpl idiom almost everywhere for the above reasons.

This pattern provides additional value for SDKs. Especially for SDKs like Qt that are intended to work on multiple operating systems. Using Pimpl to hide the implementation keeps the interfaces simple. This way they can be easily made to provide binary code compatibility. The binary interfaces are the same for every version of Windows. The same is true for every version of Linux.

Pimpl idiom is implemented by providing a class implementing the needed interface but without the actual implementation. In addition to the public functions, the class definition displays the forward definition for the class containing all class functionality. It is not necessary for the class containing the functionality to know the class providing the interface.

2.4.4. Model/View pattern

Model/View architecture is an adaptation of the Model-View-Controller architectural pattern (MVC). The main ideas behind MVC are code reusability and separation of concerns. Figure 2.4 shows a diagram of MVC architecture.

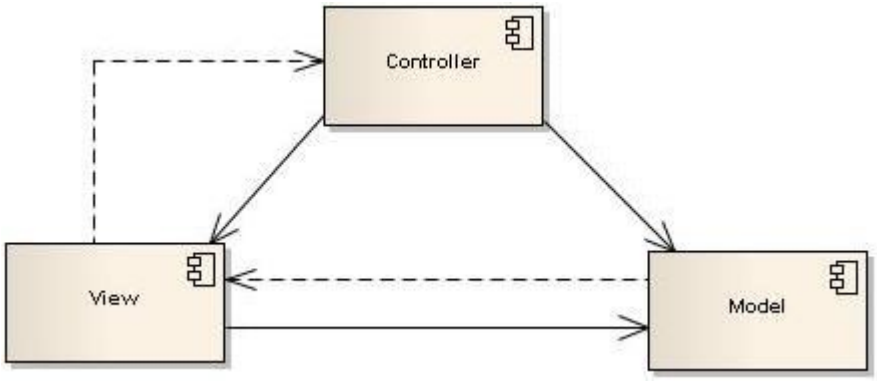

Figure 2.4. Simplified diagram of the MVC architectural pattern

The pattern isolates the application logic from the user input and view presentation. The model encapsulates the domain-specific data of the application. It also contains the domain-specific functionality operates on the model data.

The views render the model onto the screen for interaction. Multiple views can present the same model data in different ways, e.g. 3D graphics view and a view of scenegraph object properties. The user interacts with the user interface causing an event that is sent to a controller handling the view's events.

The controller receives the user input and calls suitable model operations to handle the input. When the model changes its state, it notifies its associated views so they can refresh themselves. All views listening for this type of model data change query the model for updated data. After this the views refresh themselves.

Model/view architecture is the result of combining the view and controller objects in the Model-View-Controller architecture. The resulting architecture is a simpler framework that still keeps the data storage separated from the way it is presented to the user. In the same way as in the MVC architecture, Model/view architecture allows the usage of a model in multiple views and the implementation of new views without the need to change the underlying data structures.

To provide a more flexible handling of user input, Qt introduces a concept of a delegate to the architecture. Delegates are used to customize the way data is displayed and edited in views. The views own the delegates. Figure 2.5 shows a diagram of the model/view architecture as it is used in Qt.

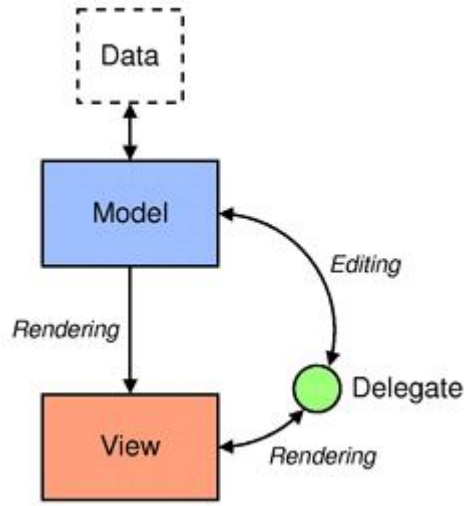

Figure 2.5. The way Qt uses model/view architecture

Qt uses the model/view architecture mainly to provide different kinds of list, table and tree views. In some cases the models can be linked to dialogs allowing a more usable interface. The different parts of the architecture communicate with each other using the signals and slots.

3. REQUIREMENTS

This chapter describes the requirements of IHA3D. Subchapters 3.1 and 3.2 discuss the main points of functional requirements along with the description of the existing system. The new system and its technical and quality requirements are described in Subchapter 3.3.

3.1. Existing system

IHA3D is a flexible and versatile visualization environment for robot simulation. It has a built-in editor which allows the user to create virtual robot models by loading mesh data files and combining them in the desired order. It makes possible for the user to configure the robot characteristics for the task. The software also allows for creation of complex robot structures such as closed structures for cylinder simulation. Created robot models and the 3D environment can be saved and loaded for later use.

IHA3D makes it possible to visualize a created robot in real-time from multiple viewpoints simultaneously. This gives the user a good overview of the environment the robot is in. The robot's characteristics and other interesting values can be shown on the screen at all times. Examples of these are the robot parts position, orientation and joint values. The Figure 3.1 shows an example of the IHA3D application GUI.

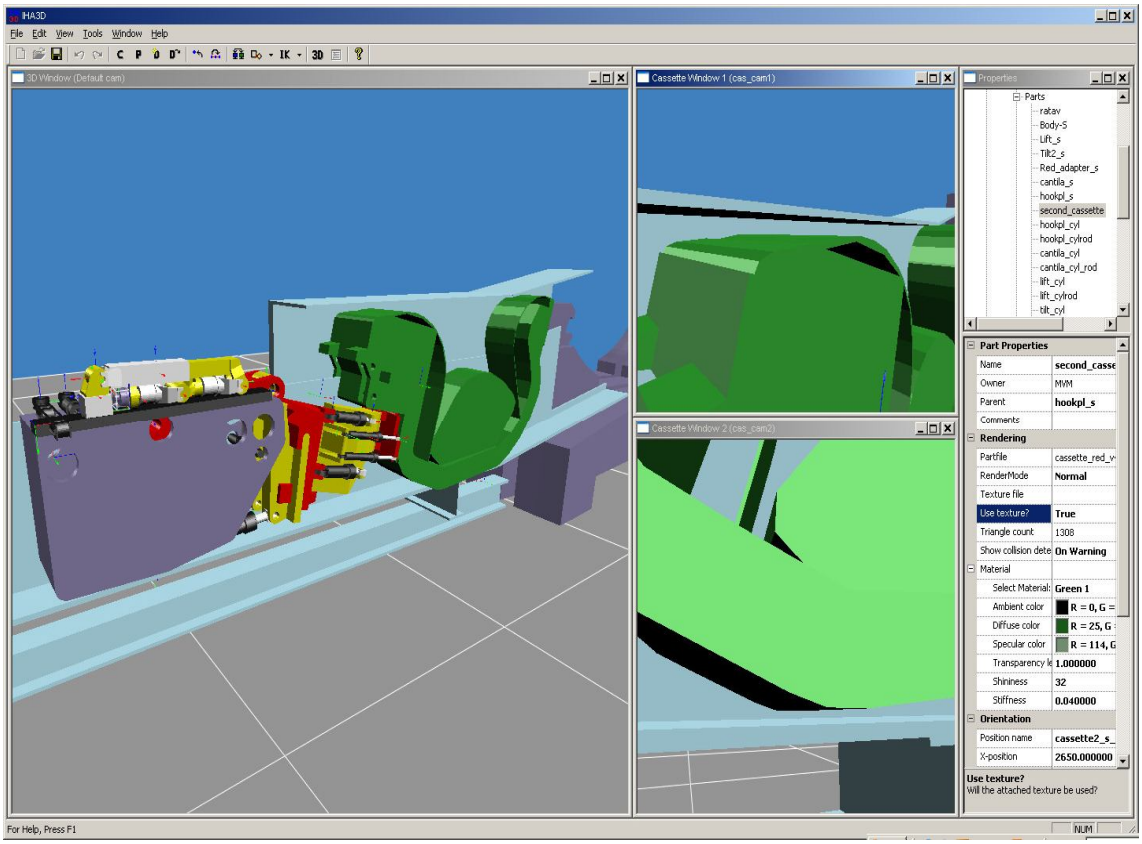

Figure 3.1. IHA3D with three windows viewing the scene

IHA3D has more features besides allowing the user to create virtual robot models. Created robot models can be controlled by several means. Firstly, IHA3D can connect to a remote server and the virtual robot can be controlled using outside data. The robot can receive joint values of a real robot, showing its position and orientation in the virtual environment. In addition to receiving data, IHA3D can also send important model information via outgoing network connections. Secondly, IHA3D provides the capability to control the virtual robot locally by various control mechanisms. One of the most useful of the local control mechanisms are the haptic devices. They provide control in three dimensions and allow the user to feel the virtual objects. The control mechanisms system of IHA3D provides interfaces for extending the application with additional control mechanism modules.

IHA3D also incorporates fast collision warning and detection algorithms. These algorithms allow the user to observe when virtual robot is about to collide or is colliding. This information can of course be selected to be shown on the screen but it can also be sent to a target computer by using outgoing network connections.

3.2. The existing scenegraph structure

Scenegraph is a data structure used to hold the object data of the virtual environment in a logical order. It consists of nodes connected in a graph structure. A node may have many children but often only a single parent. The effects applied to the parent node affect the child nodes. A good example of this is translation of a scenegraph node with many children.

The scenegraph of the existing IHA3D version was based on the Chai3D visualization library. The scenegraph objects were derived from the classes Chai3D uses in its scenegraph. This binds the data content of the model tightly to the GUI. Figure 3.2 presents a simplified diagram of a structure of the old scenegraph.

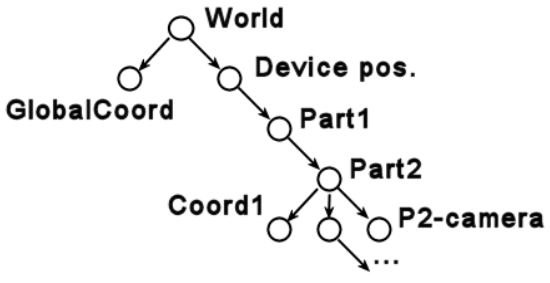

Figure 3.2. Treelike structure of the 3D world

3.3. Need for a new design

The design of the existing IHA3D was sound, but as the system evolved, difficulties crept in. This in turn caused the need for a new version. The scenegraph was too tightly bound to the Chai3D rendering engine. The model was locked when the rendering process was running. This slowed the update of model data and other functionality using the data was put on hold.

The class structure became too tightly coupled. The GUI had dependencies to the model classes. Changes to the code had widespread effects and new features needed more work. This also affected the compilation time, which was about ten minutes, for the application.

The plug-in system worked well on the previous IHA3D, but most parts of the application were not designed with extensions in mind. There was a working extension interface for control mechanisms and inverse kinematics, but other features needed to be designed and structured anew for extensions to work well.

3.3.1. Technology requirements

Redesign of the application was needed to loosen the tight coupling of different modules. The redesign also presented the opportunity to design the application with a system wide support for extensions. This decision greatly affected the architectural design as it meant that new features should be modular to make modification easier.

As the previous version of IHA3D was built using MFC classes, it was strongly dependent on MFC on more aspects than the GUI, e.g. file handling. MFC seems to be difficult to learn and makes unordinary things difficult to implement. In addition, MFC is an outdated technology that is no longer improved and rarely used. To fix this problem, it was decided to use Qt as the basis for the application GUI.

There are several features that spoke for the use of Qt in the design. It is more intuitive than MFC in its class structure and usability. It has easily implementable network functionality. There is a large community and many tutorials and examples assisting in difficult situations. Qt is also a cross-platform SDK. This makes it possible to port the new IHA3D to other platforms with minimal effort if needed.

The Chai3D rendering engine had also become outdated. It evolved too slowly and bugs were slow to be fixed. The application also needed to provide an augmented reality view of the simulated robot and its environment which seemed to be unsupported in that version of Chai3D. These issues prompted the changing of the rendering engine.

3.3.2. Quality requirements

This chapter explains the chosen quality requirements for the software. These requirements were mined from the requirements of the previous system and from the properties that needed to be improved.

Performance is an important quality attribute for this kind of software. The application should have short response times so as to feel “fast enough”. This means that functions requiring long processing times need to be in a separate thread from the GUI allowing them to run for several screen refresh cycles. The general response time was set to 33 ms according to minimum platform requirements of a single-core PC. [18]

The **modifiability** requirement is about changing the software and the cost of that change. The modifications may include improvements, adaptations for possible changes in the environment and corrections for errors or behavior. Many times modifiability is achieved using component architecture that separates different functionality and keeps the changes local. [19]

Reusability is an important requirement for productivity in the software industry [10]. Reusability is the likelihood that a segment of source code, class or software component can be used again in some other part of the software to implement different functionality with little or no modification. Reusable code reduces implementation time, prior testing has most likely removed most of the bugs and usually localizes code modifications.

Usability is important requirement for all successful applications. It is a many sided and difficult attribute to define. It can include both programmer and user oriented point of view, though it is usually thought from a user’s point of view [11]. User expects software to be easy to learn, satisfying to use and efficient to use on a particular task. Programmer expects software to be easy to modify, maintain and extend. Usability also includes the possibility to control the application with many different control devices like haptic controller or joystick.

Interoperability requirement was selected for the distributed nature of the target environment [12]. It is the ability of diverse systems to work together. It describes the capability of different programs to exchange data via a common set of exchange formats to read and write the same file formats and to use the same protocols. In this case, the application is implemented with the ability to send and receive a selected set of commands defined by the environment used to set the position and orientation of a robot.

Availability requirement was selected because it is expected for the application to be usable for extended periods of time [13]. Availability is the ratio of the total time a system is capable of being used during a given interval of time. The system is expected to stay online at least a day without problems as no-one works on the project at night.

Extensibility is important as not all the functional requirements were known at the start of the development and more could surface even after the project has ended [14]. Extensibility measures the ability to extend a system and the level of effort required to implement the extension.

4. DESIGN

First, this chapter explains the environment of the system. Then, the general structure of the software is explained. After that the architecture is examined from the PLA point of view ([1], p. 157-186). Next an overall view of the modules implemented in the executable is given.

4.1. Environment

The software is implemented for a PC platform. The required operating system is Windows XP or newer. It is designed primarily for use on a multi-core processor hardware with a modern (year 2004 or newer) graphics card for 3D graphics. Remote operation control data reception requires a network connection for server access. Different plug-ins may have additional demands for hardware.

4.2. Overview of the architecture

Figure 4.1 shows the high level structure of a plug-in based system. The application consists of two larger modules that make use of the smaller ones. One is the Core that handles model related functionality. The other is the IHA3D_Application that provides the GUI for the application and integrates the Core with the GUI functionality into one executable application. The plug-in types are specified more thoroughly in chapter 4.8.

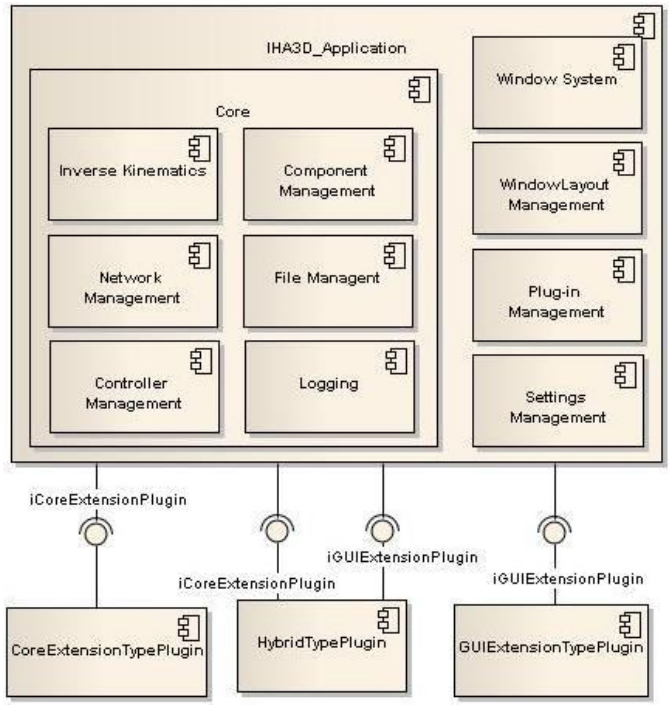

Figure 4.1. The architecture at a high level

The architecture of IHA3D consists of two architectural patterns. The *IHA3D_Application* module uses Model/View pattern for providing the GUI. Model/View architecture pattern is very much like the MVC-pattern, but with minor changes as described in Chapter 2.4.4. It is used to separate the data storage from the way the data is presented to the user, but provides a simpler design than the MVC pattern. This separation of data and view allows multiple kinds of views for the same data model.

The core is mainly a message dispatcher architecture pattern described in Chapter 2.4.2. In addition, it also provides services of the Core modules shown in Figure 4.1. These services can be extended via interfaces that the core provides for plug-ins. The Figure 2.2 shows an example of a message dispatcher pattern.

4.3. Product-line architecture viewpoint

The PLA can be viewed in a layered style to examine its structure in a hierarchical manner. The model can usually be found from the PLA even when it was not deliberately built that way. Figure 4.2 displays a layered model of a PLA in a generalized manner. The figure shows the model has four layers that make the whole. ([1], p. 175)

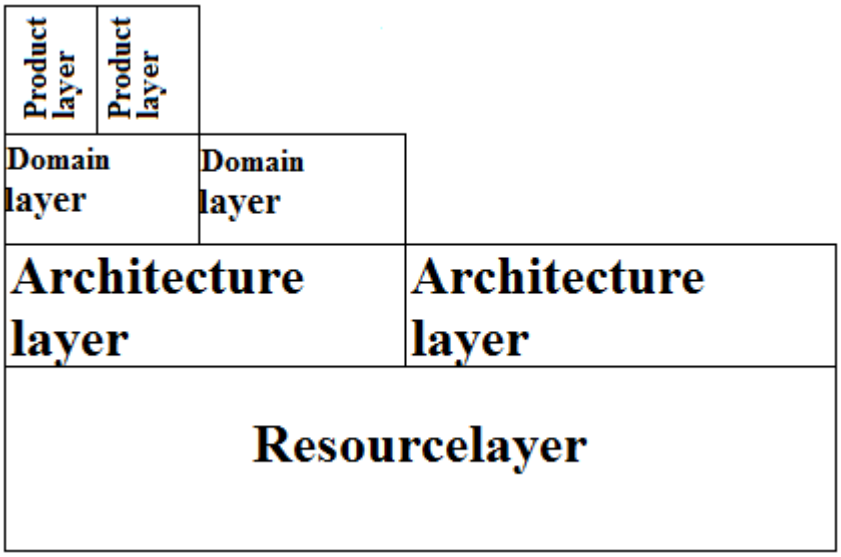

Figure 4.2. Layered model of PLA

The lowest layer contains the management functionality that hides the general platform dependent services and resources. It provides interfaces for the rest of the application to access the hidden functionality. Depending on the platform, these may for example be communication, process handling or memory management related. In this case, there are libraries and their APIs containing compatible data types and functionality. They are meant for easier and more consistent development of plug-ins and further improvement of the executable. ([1], p. 176)

The architecture layer defines the general architectural style for the software. It offers attachment points for the application related layers. This PLA's architecture is composed of Model/View, Dispatcher and Plug-in architectural patterns. In this PLA, this layer implements the plug-in management interfaces as the main variation point. These are discussed more deeply in following chapters. ([1], p. 177)

The domain layer builds upon the general architecture defined in the architecture layer. It implements the frame for the applications on their specific domain and may contain additional architectural solutions. In this case, the interfaces some components provide are considered to be additional, secondary, variation points. New components and interfaces are added to the design as deemed necessary when evolving the system. ([1], p. 177-178)

The product layer contains all the applications developed using the PLA. This layer implements the product specific functional requirements for example, GUI interfaces for user to interact with plug-ins if necessary. ([1], p. 178)

4.4. Plug-in management

Plug-in management software module is a fundamental part of the functionality of an extendable application. The application uses Qt SDK to implement the plug-in management system. Qt has two ways of loading a plug-in. One loads Qt defined plug-ins and the other loads plug-ins implementing low level exported C-style interfaces. The first interface type is used to add more functionality to the application's GUI. The second is used to extend the model data handling of the Core component.

The Core plug-in interface has no dependencies to third-party libraries like Qt or Boost. This gives the plug-in developer more freedom in the plug-ins' internal design, though there are some restrictions. The GUI extending plug-in interface forces the use of Qt SDK in the plug-in. This choice was made to ease the handling of GUI elements. All computationally intensive processing should be kept separated from the main thread handling GUI elements to keep the user interface responsive. This problem is already solved in the Qt SDK. The signals and slots system of Qt handles the transfer of computation from processing thread to the GUI thread automatically. This simplifies the design greatly.

Plug-ins may have different lifetimes, but all of them follow the same lifecycle. The cycle has five steps that are

- loading,
- initialization,
- execution,
- uninitialization,
- unloading.

First, a DLL file is loaded into the memory to check if it implements at least one of the accepted plug-in interfaces. If no interface is found the file is unloaded from the memory. The plug-in's general data, like its description, name and GUID, are read when the plug-in interface is loaded and accepted. When the plug-in is loaded into the application, it can be initialized using the implementation of the plug-in interface. Initialization instantiates the plug-in's functionality allowing it to use the interfaces the application offers for new components. While the plug-in is in execution phase, it can function as a part of the application. The plug-in is uninitialized when it's no longer needed. Uninitialization removes all components the plug-in has added to the application after which it cannot use any of the services the application offers. The plug-in remains in memory after the uninitialization and can be initialized again. When the plug-in is unloaded, it's removed from the application memory. If it's needed again, it must be loaded again for initialization to be possible.

4.5. Visualization

The 3D rendering is separated in to its own extension plug-in. It extends both the core to get all the scenegraph related events and the GUI to add a rendering window and other functionality for the user's convenience. The interaction between the application and the plug-in is explained in detail in the plug-in management chapters. The visualization component implements the Dispatcher pattern to receive and handle events from other parts of the application.

4.5.1. Scenegraph

Previously the scenegraph was strongly tied to the Chai3D implementation and was part of the rendering process. The scenegraph needed to be separated from the rendering engine to avoid locking the model as it is rendered. This would speed up the update time of the model data and have a lesser impact on the frame rate of the rendering engine as neither thread needs to wait the other so long. Figure 3.2 shows an example of the old scenegraph.

The scenegraph‘s structure also needed to be changed to implement new features. The old scenegraph didn't allow for creation of global parts, like walls, into the virtual environment. There was a need for calculating the position and orientation of two objects in the scene in relation to each other.

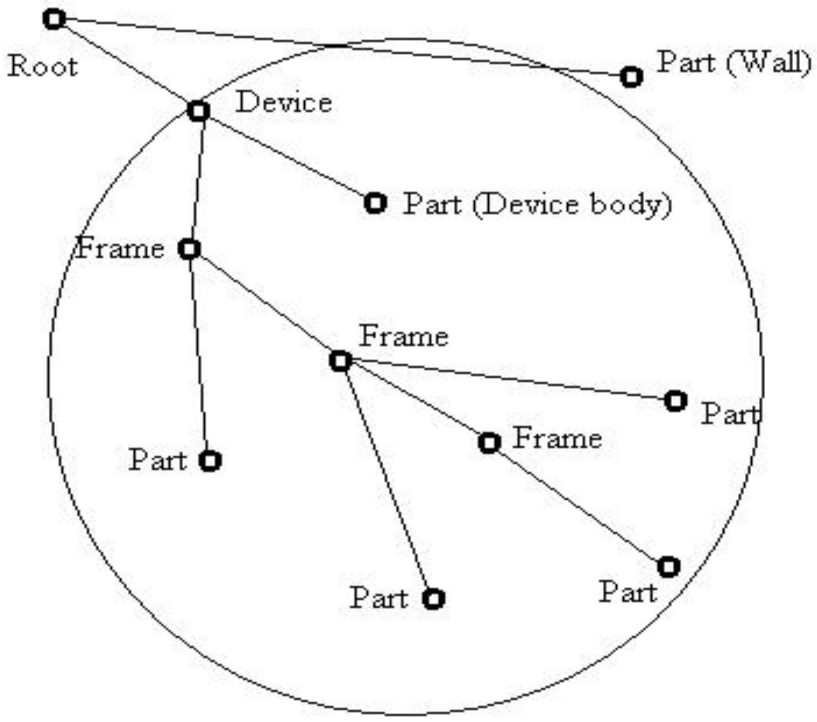

Figure 4.3. Example scenegraph structure

The old scenegraph was limited in that it enabled only one child object to a parent coordinate. This limited the device structure. It required more calculation to make a device with many branches to seem whole. The new scenegraph, shown in Figure 4.3, was designed so it enabled attaching many objects as children to the same frame coordinate. These Frames attached in the tree structure constitute the framework of the devices in the virtual world. The meshes displaying the visible parts of the device are attached to the same Frame. This allows several different Parts and Frames to move with their parent.

4.6. Module design

4.6.1. Core module

The Core holds the application's functionality together. It controls the creation, lifetime and access of all the software components. Its modules handle the registration of new components defined in plug-ins that want to have access to the model data. It ensures the components handling the model are kept up-to-date. All the changes to the model data pass through the core's component management module which handles internal communication of the Core.

The Core provides many services to plug-ins. Figure 4.4 shows the core's internal structure and the interfaces it offers to loaded plug-in components. The components also offer interfaces to plug-ins, but these are described in the components own chapters.

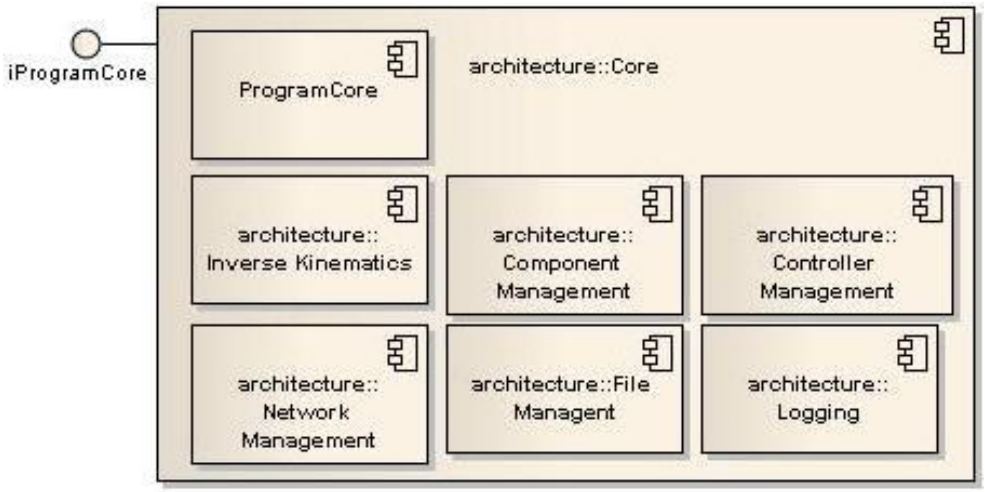

Figure 4.4. Core component's internal structure

Core's access point and focus is the ProgramCore component. It is the first component created in the core and controls the lifetime of the other parts of the core. Plug-ins can access the ProgramCore through iCorePluginAccessPoint interface and core other components other interfaces via ProgramCore's implementation of iProgramCore interface.

4.6.1.1 Core interfaces

The Core offers interfaces to the rest of the application. One of them is accessible also for plug-in. The others are used for configuration purposes within the application. Figure 4.5 below shows the interfaces visible outside of the *ProgramCore* component. The description here is not enough to get a good view of the *ProgramCore* component. A better understanding of the *ProgramCore*'s functionality and modification can be obtained by examining the sequence diagrams detailing the components execution.

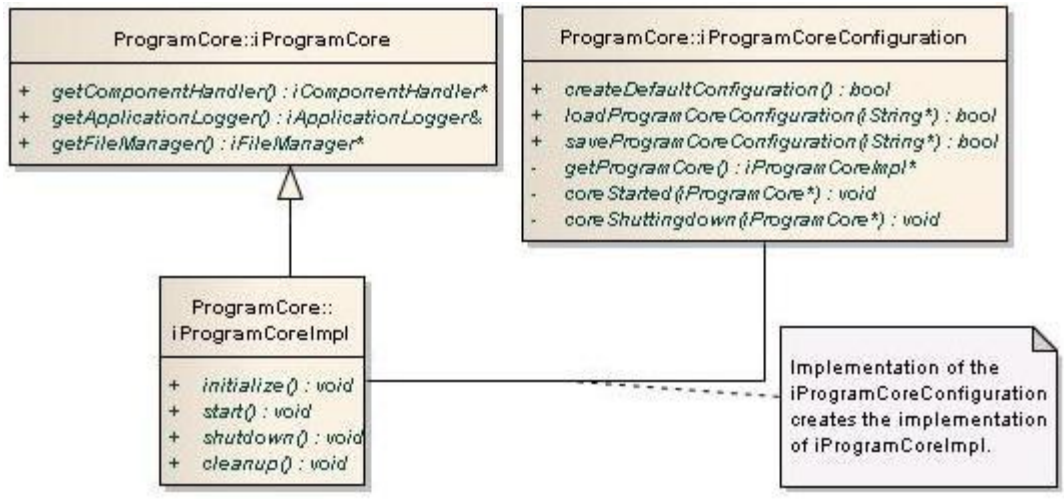

Figure 4.5 Externally visible interfaces of the ProgramCore component

The *iProgramCore* interface is the access point to the *ProgramCore* component. It is given to the plug-ins to integrate them as a part of the application. It presents methods to access other components of the Core. The *iProgramCore* uses pimpl idiom to hide changes in the core from other modules in the executable.

The *iProgramCore* and *iProgramCoreConfiguration* interfaces are intended for configuration the application's functionality. By creating different implementations of the *iProgramCoreImpl* interface the program can be used for different purposes. For example, an operator may need the networking capabilities, but a designer only needs the model modification capabilities.

The *iProgramCoreImpl* interface defines the internally needed methods of this component. It is used to handle the lifetime and execution of the entire Core component. After start()-method call the core is ready to be used. Calling shutdown()-method in turn makes the core unsafe to use outside the shutdown process.

The *iProgramCoreConfiguration* interface defines the actions needed for configuring the Core. It creates the implementation of *iProgramCoreImpl* interface defined by the configuration settings and manages its lifetime.

4.6.1.2 Component management

Component management module in IHA3D provides interfaces for extending the model handling functionality of the Core. This module is central to implementing the message-passing pattern. Two interfaces are needed to add new components to the Core. One of them is presented in the Figure 4.6, which shows interfaces the component management exposes outside.

This *iComponentHandler* interface is used to manage the registration of new components handling the model data. The component is thread safe and can handle registration and removal of asynchronous core extension components. The interface, new components need to implement, is shown in Figure 4.7. The *iCommunicationHandler* interface is explained better in Chapter 4.6.1.3.

The *iComponentHandler* interface also provides the possibility to use a registered component directly by acquiring its interface. This connection demands that the GUID of the components is known beforehand. Direct contact between components helps speed up communication between components that are specifically designed to work together. It should be noted that a component can't be removed until all acquired references to it have been released.

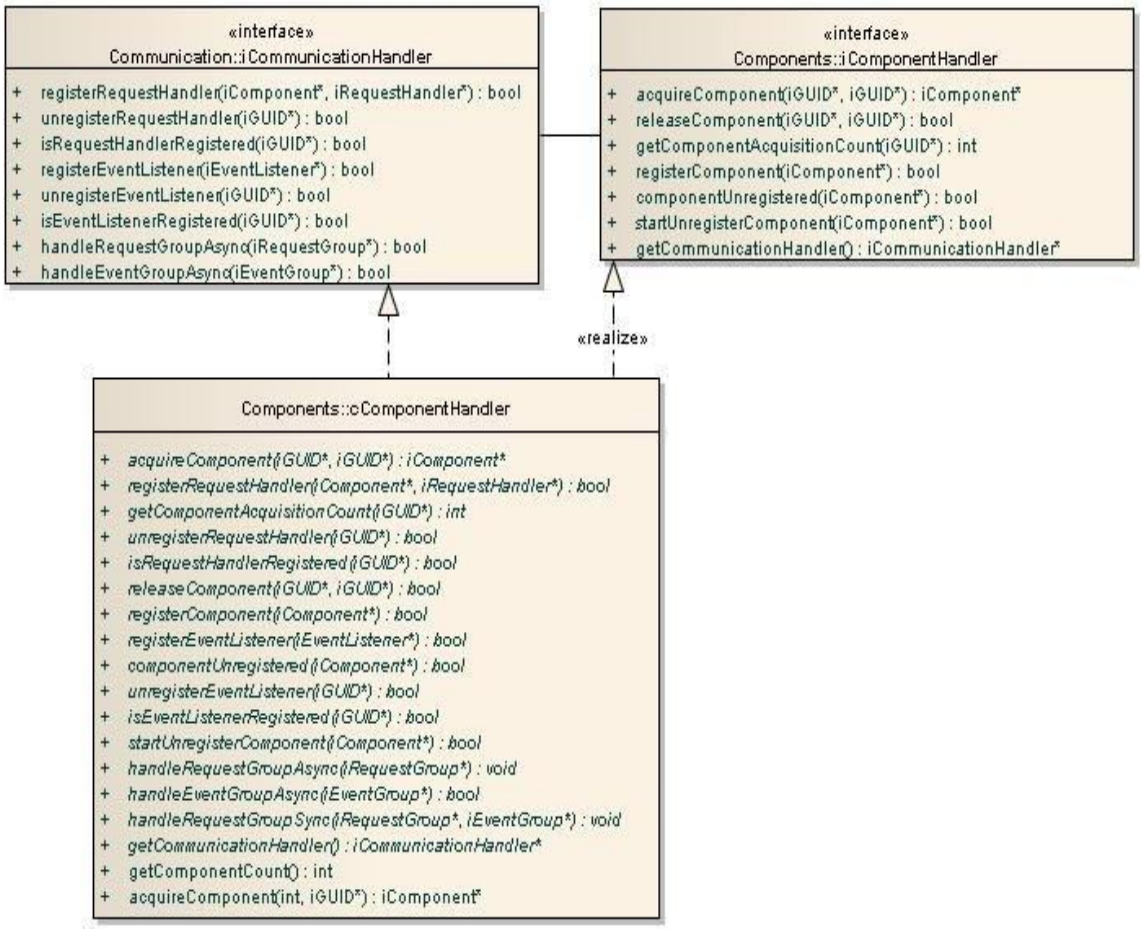

Figure 4.6. Exposed interface of the component management

The *cComponentHandler* class implements the functionality defined in the interfaces of Figure 4.6. It is used by the functionality of the executable. Functionality implemented within a plug-in can only use the interface definitions.

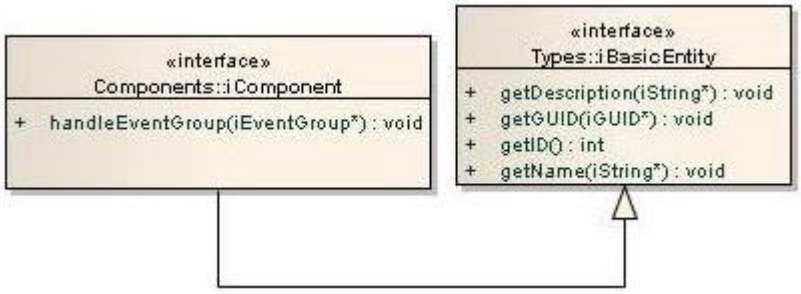

Figure 4.7. Interface component management reguires from components handling model data

New components extending the Core's model handling functionality implement the *iComponent*. This interface enables the new component to receive events of model data changes from the communication handler. It inherits *iBasicEntity* interface which provides basic functionality implemented by many objects in the software. It requires the component to have its own identifiers and description.

The handling of events may be synchronous or asynchronous, but it is advisable to place computationally intensive processing in a separate thread. This keeps the communication thread light weight and responsive.

4.6.1.3 Component communication handling

IHA3D defines many events for handling the model data. Their types are identified with unique GUIDs. Developers can define their own event types as needed and use them in extension components.

Component management module receives the events and distributes them on to interested listeners and components. It reveals one interface to the outside which is presented in Figure 4.8 and it uses three others presented in Figure 4.9 excluding the iComponent interface shown earlier in Figure 4.7.

Figure 4.8. Interface that the Communication handler component reveals

Component management module implements the iCommunicationHandler interface to manage. It is prudent to keep the registered components and event passing close to each other as communication handler must send the events to all components. The iCommunicationHandler interface handles the event passing to the listening registered components.

The Figure 4.9 shows the interfaces that are used with events. The iEventGroup and iMessage interfaces contain the information passed around to all interested handler implementations. They are described in the Appendix A in more detail.

The *iEventListener* interface is for light weight single event handling. The listener is mostly used in component's internal event handling when parsing an event group. The *iCommunicationHandler* provides an interface to register listeners for a system wide listening.

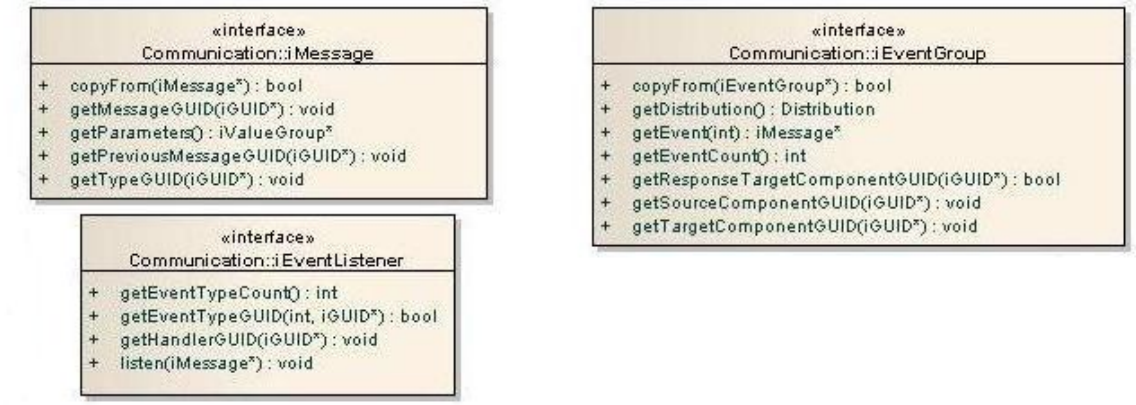

Figure 4.9. Interfaces the Communication handler uses to pass messages to components

Event handling

Every instance of iComponent interface implements its own event handling. Figure 4.10 shows an event handling sequence where a ComponentA changes the data and informs other components of the change.

Events are always sent to the CommunicationHandler that will set them to a buffer to be distributed later. The event distribution operation is usually done asynchronously and thus allowing the sender to continue its own task. The events are sent to the registered components by calling the event handling method of the iComponent interface of each component. The rest of the event handling sequence depends on the implementation of the component.

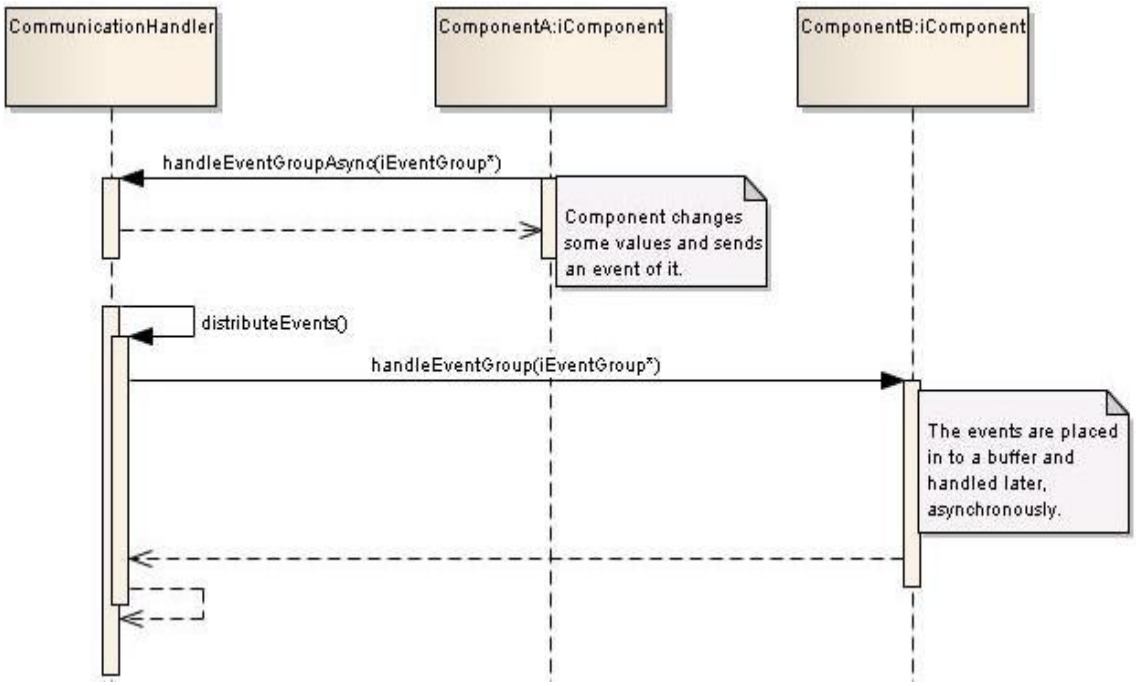

Figure 4.10. Example event handling sequence

4.6.1.4 Network management

The network management module manages the network protocols used by the application. It keeps track of the ports the application uses as they are a limited resource in any system. The module provides interfaces for registering new protocols and using them. The networking is implemented so that only the user using provided dialogs or the implementation of the program executable can authorize network connections using this module.

The network protocols are based on the commonly used UDP protocol. The UDP protocol is faster as it has no need to connect to the target server. It also does not require the sender to wait for the confirmation of successful transmission of the network packet.

Provided interfaces

The Network management module exposes three interfaces outside the module which are shown in Figure 4.11. Only the *iNetworkManager* and *iUDPPacket* interfaces are relayed to the plug-ins implementing the packet handlers. The *cNetworkManager* class implements the main functionality of the module and thus is exposed only to the implementation of the program executable.

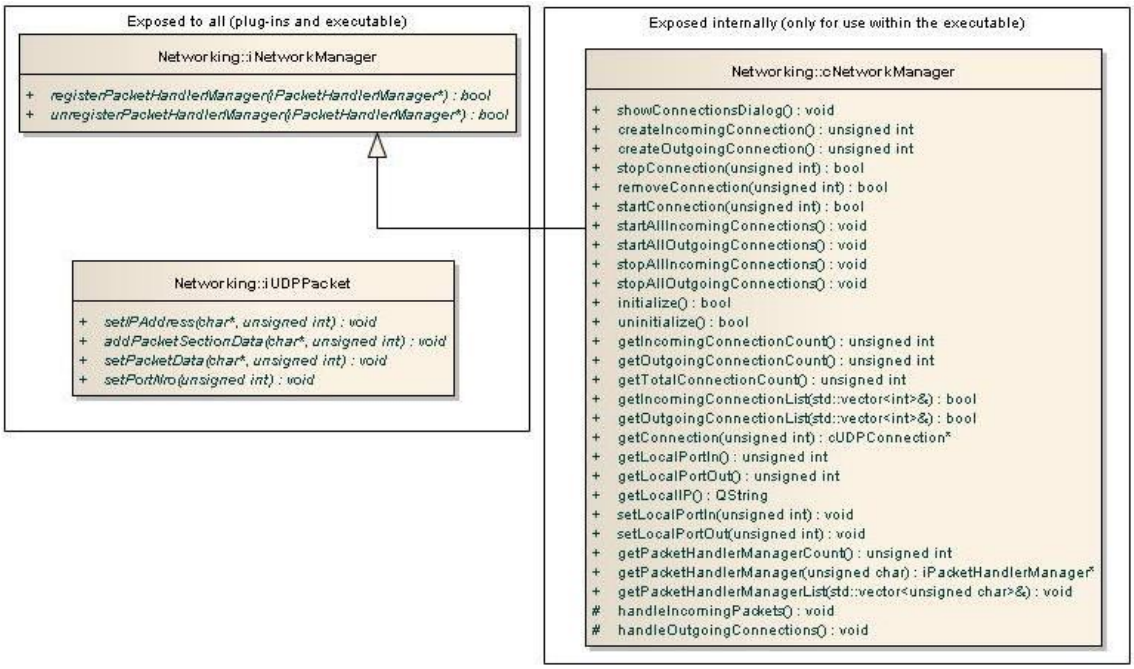

Figure 4.11. Interfaces of the network management module

The *iNetworkManager* interface is used to manage the registration of new types of network packet handlers to the system. In itself it has no implementation as it is designed to cross the DLL boundary safely. It is implemented by the *cNetworkManager* class which is the main class in the module. It controls the use of network packet handlers as it directs all network traffic. The incoming network packets are relayed to

their assigned handlers as soon as they are received. Rest is left up to the plug-in implementing the packet handler.

The *cNetworkManager* contains a timer to space the sending of outgoing network packets. The user sets the interval of time between two consecutive outgoing transmissions. This is meant to prevent the network from clogging up with excess packets. Also there rarely is a need to send data at such a rapid pace. The outgoing network packets are filled just before sending to ensure they contain most resent information.

Figure 4.11 presents the *iUDPPacket* interface. It is used to transfer packet data between the handlers and the *cNetworkManager*. It provides a way for the handler to set the data in the packet or just append more after the existing data.

Used interfaces

The network management module uses packet handling implementations provided by plug-ins. These packet handlers must implement the interfaces presented in the Figure 4.12. Plug-ins implementing network packet handling features have four interfaces to implement.

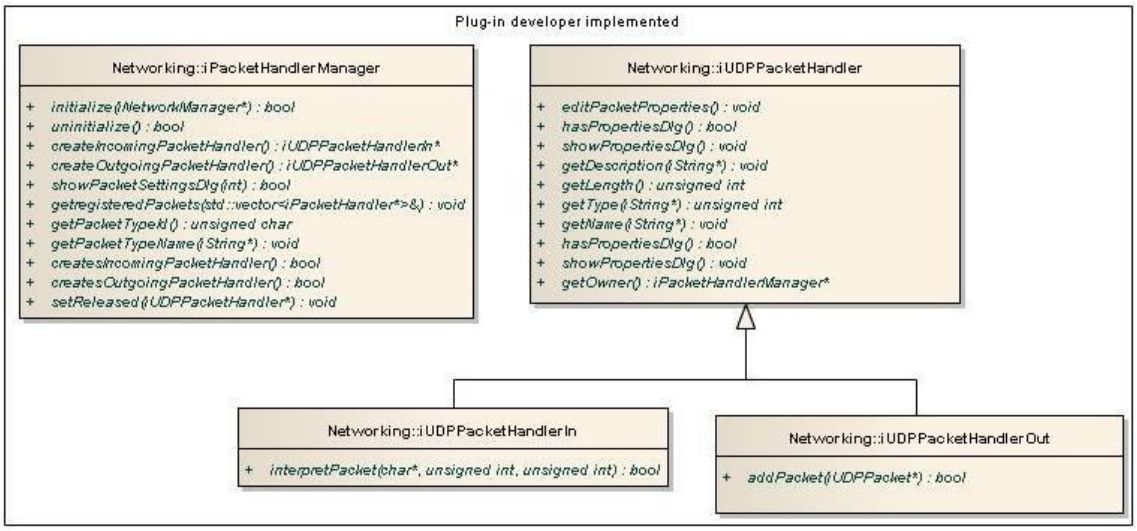

Figure 4.12. Interfaces implemented in plug-ins

The network management module uses the *iPacketHandlerManager* interface is a factory object used to when creating a new packet handler. Plug-in registers the object implementing the interface using *iNetworkManager* interface provided by the executable. The *iPacketHandlerManager* provides information as to what type of packet handlers it can create and other helpful data for identification. Network management module uses its interface to build a list of available packet handler types. The *iPacketHandlerManager* creates and deletes the packet handlers of its type. It

requires the application to inform it when releasing a packet handler when the handler is no longer needed.

The *iUDPPacketHandler* interface is the base class for all network packet handlers. There is one packet handler for each connection. The handler implementation holds the information as to what the network packet it handles contains. It allows the network management module to access the handler's basic information identifying it. It provides the interface to query if the handler has a dialog for editing the packet content controlled by this handler. Handler is responsible for displaying the dialog as it has to easily access the handler's data and it would be harder across DLL boundary.

There are two types of packet handlers. Network packet handler implements either *iUDPPacketHandlerIn* or *iUDPPacketHandlerIn* interface to gain access to the network. The *iUDPPacketHandlerIn* defines a function for interpreting the incoming data. The handler uses this data to change the scenegraph to correspond with outside state, or to affect the functionality of the application. The *iUDPPacketHandlerOut* defines a function for filling a provided network packet with new data. The network management module calls this function when it is time to send another packet of the handler's type.

4.7. GUI module

The IHA3D_Application module is the main module of IHA3D. It handles the creation of all other modules in the application. It initializes and manages the GUI and provides an extendable user interface for the user.

4.7.1. API for extending IHA3D GUI

IHA3D provides an API for extending the application's GUI. The API consists of two different parts, the first part being obligatory if the GUI is to be extended. The obligatory interfaces are covered in section 4.7.1.1. The second part contains optional interfaces which provide additional extension possibilities. These interfaces are covered in section 4.6.1.2.

4.7.1.1 Obligatory interfaces

The purpose of the obligatory interfaces is to provide a common way of accessing the IHA3D GUI for all GUI extension plug-ins. Figure 4.13 shows a class diagram depicting these interfaces.

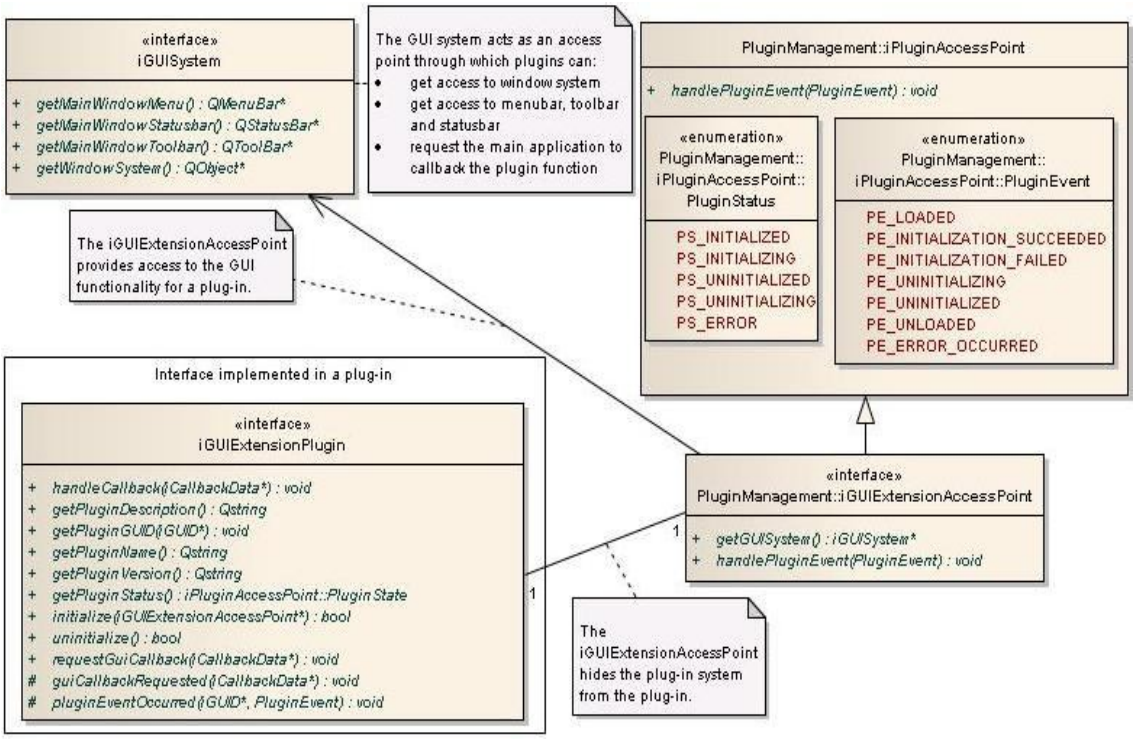

Figure 4.13. The obligatory interfaces for extending IHA3D GUI

The obligatory interfaces consist of four different interfaces (Figure 4.13). The iGUISystem interface is implemented on the main application side and it will be given to those plug-ins that want to extend IHA3D GUI. The main purpose of the iGUISystem is to act as a point for GUI extension plug-ins so that the plug-ins have a common interface from which they can access various GUI elements. For example iGUISystem can be used to get access to window handling (WindowSystem), main window menus, statusbar or toolbar.

WindowSystem is one of the most important and most used possibilities for extending the IHA3D GUI. It allows registering new window types (e.g. 3D-windows or Augmented Reality windows). As said, WindowSystem can be accessed using the getWindowSystem() function in iGUISystem interface. The reason why the function returns pointer to QObject instead of the iGUIExtensionPlugin interface, is that WindowSystem has a couple of Qt signals and these cannot be connected to slots without a pointer to a QObject object. The returned QObject is guaranteed to implement the iWindowSystem interface. More information and the more detailed description of the WindowSystem is given in section 4.4.

Second interface shown on the left side in Figure 4.13 is the iGUIExtensionPlugin interface, which must be implemented by all plug-ins that want to extend IHA3D GUI. The iGUIExtensionPlugin interface contains functions related to four different areas. The first functions of the first area allow querying the plug-in properties. The properties that can be queried are shown in Table 4.1.

Table 4.1: Plug-in properties that can be queried via iGUIExtensionPlugin interface

Property	Function	Description
Name	getPluginName()	Gets the plugin name as a QString.
Description	getPluginDescription()	Gets the plugin description as a QString.
Status	getPluginStatus()	Gets the plugin status. Possible plugin statuses are uninitialized, initialized and error condition as given in in *PluginStatus* enumeration. When plug-ins are loaded they are uninitialized. After they have successfully been initialized the plug-in statuses are updated accordingly. If a plug-in faces a serious error, it is possible that plug-in may change its status to error condition.
GUID	getPluginGUID()	Gets the plug-in GUID. Each plug-in must have an unique GUID in order to correctly identifiy the plug-in. The GUID must remain same as long as the plug-in is loaded.

The second area is related to initializing and uninitializing the plug-in. Every plug-in will be initialized by calling the initialize(iGUISystem*) function before the plug-in will become operational. The iGUISystem interface is given as a parameter so that plug-ins can access GUI elements afterwards.

The third area consists of two functions: guiCallbackRequested(), which is a Qt signal, and handleCallback(). The purpose of these functions is to allow executing the plug-in code in GUI thread since GUI objects are not allowed to be modified from any other thread. So whenever plug-in emits the guiCallbackRequested() signal, the GUI thread will perform a callback using the handleCallback() function. The callback will be asynchronous.

The fourth area contains only one function: pluginEventOccurred(), which is a Qt signal. The purpose of this function is to have the plug-ins report their events to interested parties. Plug-ins are obliged to send these events whenever they occur.

The right side of Figure 4.13 presents the interfaces implemented on the application side that must be used by a plug-in extending the GUI. IPluginAccessPoint defines plug-in states and state changes. They are used inform the application of the state changes of the plug-in. This is done calling the handlePluginEvent function in the iGUIExtensionAccessPoint interface. This interface is given to the plug-in when it registers to the system.

4.7.1.2 Optional interfaces, window handling

The IHA3D GUI can further be extended using optional interfaces. The most important optional interfaces are related to window handling (WindowSystem) by providing a support for 3rd party window types. Window types, e.g. 3D-window and Augmented

Reality window, provide a way of creating custom windows in IHA3D. Figure 4.14 shows a class diagram of window handling related interfaces.

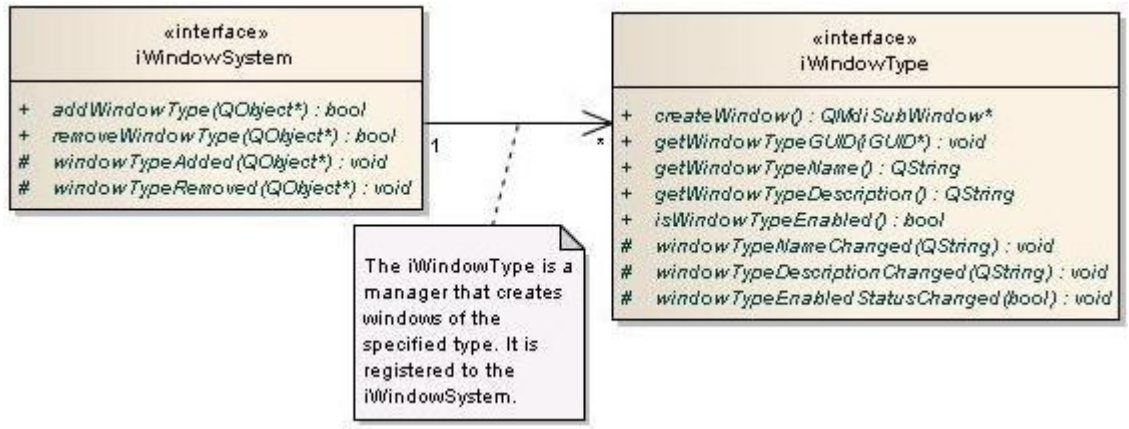

Figure 4.14. The window system related interfaces that can be used to extend IHA3D GUI

The window handling related interfaces actually contain two interfaces: *iWindowSystem* and *iWindowType*. The *iWindowSystem* interface is guaranteed to be implemented on the main application and it can be queried from the *iGUISystem* interface (covered in Section 4.7.1.1). The purpose of the *iWindowSystem* is to handle custom window types defined and implemented in plug-ins.

The *iWindowSystem* interface consists of two parts. The first part handles adding and removing custom window types. The second part lets interested parties know when a specific window type is added or removed. Each of these functions takes a QObject pointer as a parameter. The reason for this is that iWindowType interface contains a couple of Qt signals, which cannot be connected without QObject pointer. The addWindowType() and removeWindowType() functions therefore will fail if the given QObject parameters do not actually implement the *iWindowType* interface. The windowTypeAdded() and windowTypeRemoved() signals, however, are quaranteed to pass *iWindowType* as a parameter.

The window type specific information is capsulated in the *iWindowType* interface. The purpose of the interface is to let 3rd party plug-ins register their own window types to IHA3D. The implementation of the interface is therefore left to the plug-in developer.

The *iWindowType* interface has three important parts. The first part of the functionality is to give access to window type properties such as name, description and enabled status. The enabled status tells whether the window type is currently available and windows of that type can be created. For example, a window type may restrict the number of certain windows. In this case, the enabled status can be used to limit window creation. Additionally each window type must have a unique identifier, which can be queried using getWindowTypeGUID() function. The window type GUID will be copied into the given parameter.

The second interface part covers creating new custom windows, which is achieved using createWindow() function. The function returns a QMdiSubWindow pointer, but in most cases the created window will be inherited from the QMdiSubWindow instead of actually being one.

The last part of the interface consists of a couple of Qt signals. Their purpose is to let interested parties know about changes in the window type.

4.8. Plug-in types

There are two main types of plug-ins in IHA3D. The first is a plug-in extending the Core and through it gets access to the model data. The second is a plug-in extending the GUI side of the application adding elements visible to the user. There can also be a hybrid plug-in implementing both extension interfaces.

4.8.1. Plug-in interfacing

There are two interfaces with which a plug-in can register to IHA3D. Plug-in implements *corePluginInterface* interface and is registered to the core. Plug-in implements a Qt derived *iGUIExtensionPlugin* interface and is registered to the application's GUI side.

IHA3D checks first if a new plug-in implements the Qt plug-in interface. After this it checks the existence of a component interface for extending the core. If neither interface is implemented in the DLL, the file is not an accepted plug-in type. Loaded plug-in gets an interface with which to inform the application of changes in its state and can access the applications other service interfaces.

4.8.2. Core type extension component

This type of plug-in is used to add new functionality to IHA3D via the Core module. Core extension plug-ins implement the *corePluginInterface* interface as a set of exported C-style functions as to cross the DLL boundary. The plug-in is initialized, verified and un-initialized using these interface functions. The interface doesn't force any dependencies to third-party libraries. This allows for more freedom in the component design.

The plug-in gets an interface with which to inform the application of the changes in the plug-in's state. The plug-in gets access to the Core through the same interface.

The model data handling is extended by registering new components to the core module. The components are required to implement the *iComponent* interface, but the component's internal implementation is left to the component's developer to decide. The component implementations handle their own memory management, releasing the memory they reserve.

Figure 4.15 shows an example of interfaces between a core extension plug-in and the application. The necessary interfaces are presented as connected because they exist for the duration of the extension's lifetime.

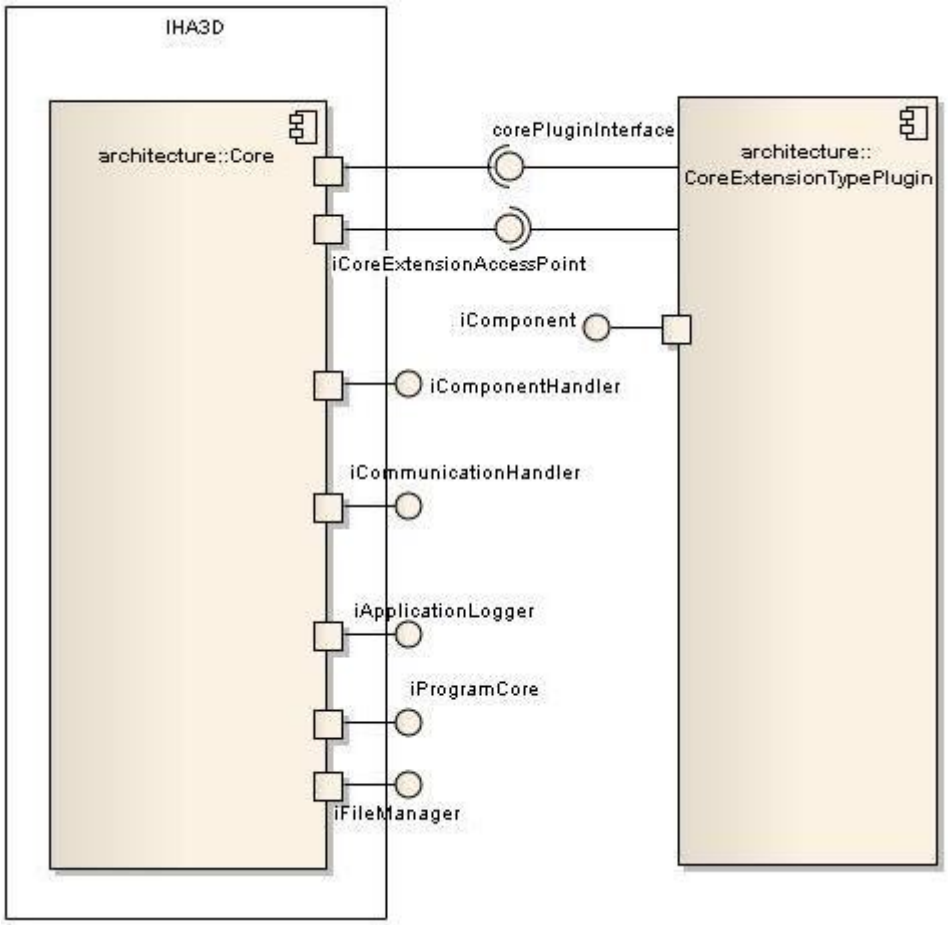

Figure 4.15. Example of the interfaces between the core and its extension component

4.8.3. GUI type extension component

This type of plug-in is used to add functionality to the application's GUI. Plug-ins extending the user interface are required to implement the *iGUIExtensionPlugin* interface. It has the same functionality as the core extension interface has with one exception. It gives the plug-in an access to the *iGUISystem* interface. This enables the plug-in to add its user elements to the application. The interface forces the plug-in to use Qt libraries as the application's GUI is based on it.

Figure 4.16 shows an example of interfaces between a core extension plug-in and the application. The necessary interfaces are presented as connected because they exist for the duration of the extension's lifetime.

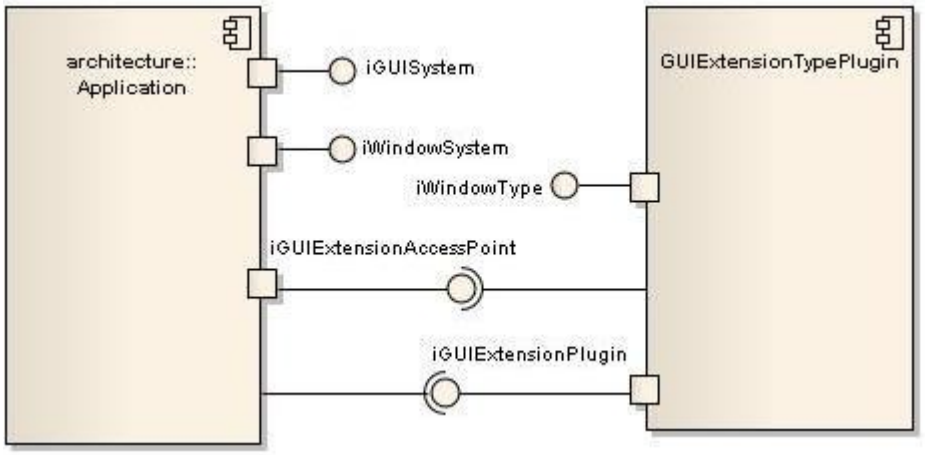

Figure 4.16. Interfaces between the application's GUI and its extension component

4.8.4. Component extending GUI and Core

Figure 4.17 shows a plug-in implementing both extension interfaces.

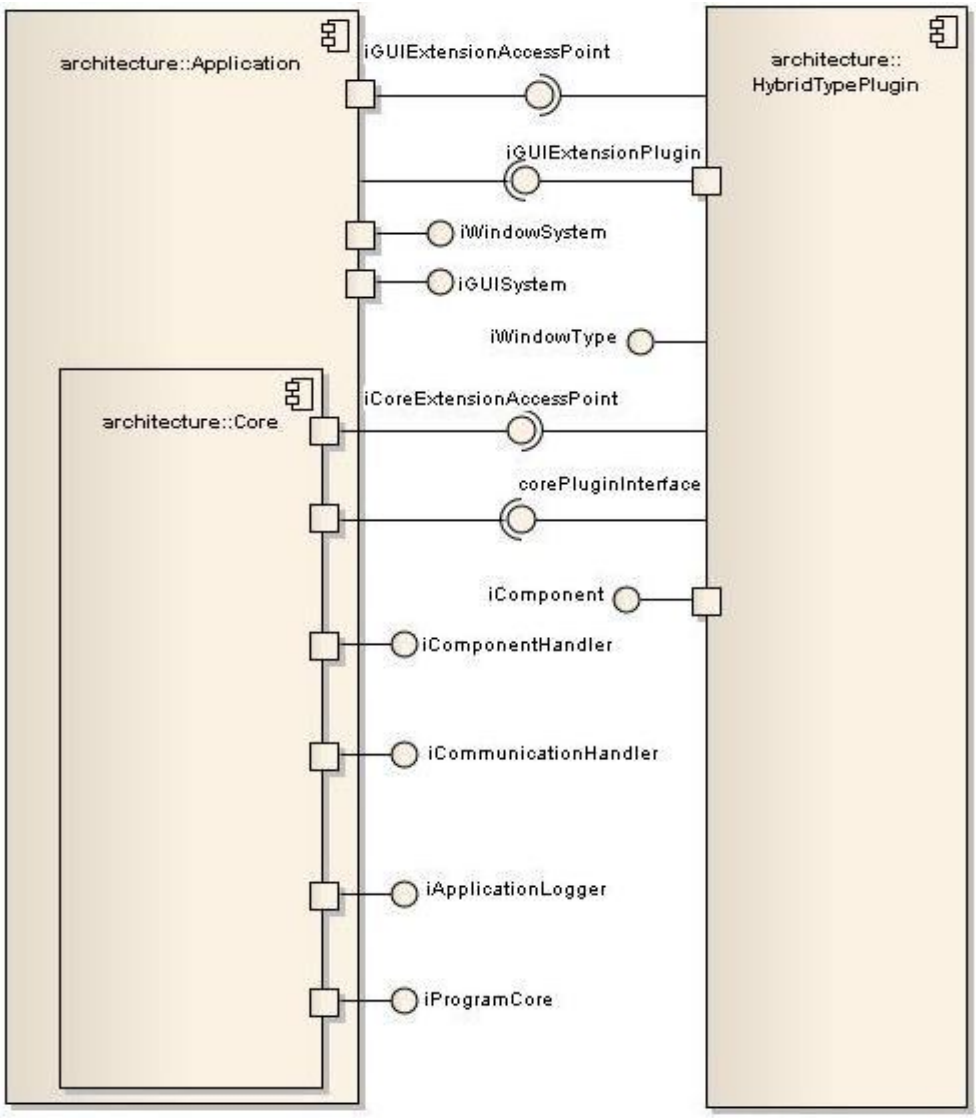

Figure 4.17. Component extending the Core and IHA3D_Application

This type of plug-in is used to add new functionality to IHA3D via both the Core and GUI of the application. This type of plug-in implements both the *corePluginInterface* interface and some of the Qt derived *iGUIExtensionPlugin* interface. This makes it possible for a plug-in to implement its own configuration and options dialogs for the functionality it adds to the Core.

4.8.5. Plug-in management interfaces to application

The plug-in management component is loosely coupled to the rest of the system. It exposes only two interfaces to the rest of the application. These are *cPluginManager* and *cPluginData* shown in Figure 4.18.

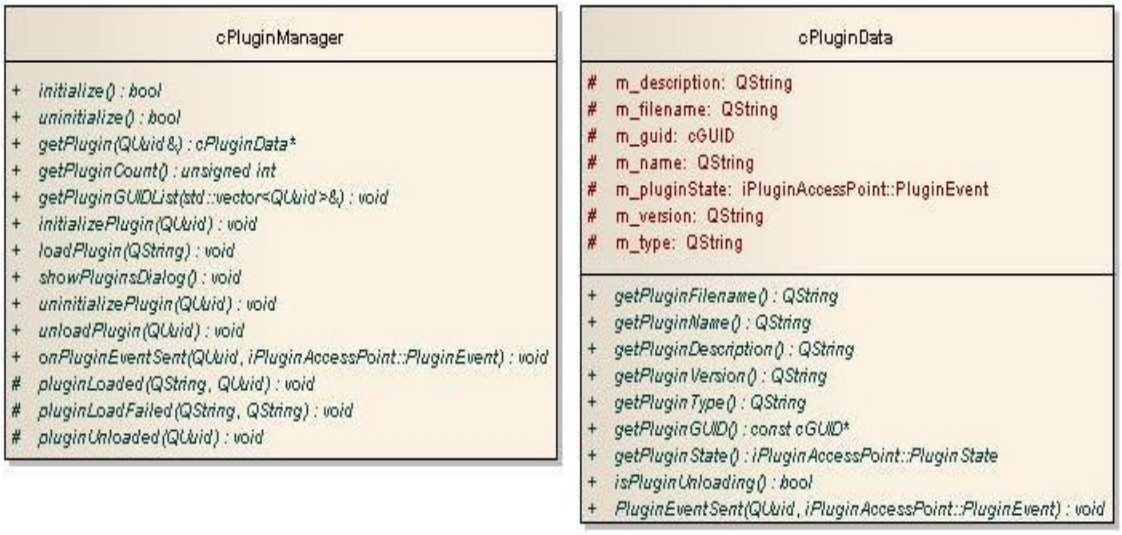

Figure 4.18. Interfaces exposed to the application.

The *cPluginData* class is used to store information of a loaded plug-in. There is one for each loaded plug-in interface. Whenever another part of the application requires information of a plug-in, it is given *cPluginData* object. This prevents rest of the application from having direct knowledge of the plug-ins. The interface offers the possibility to get information of a plug-in, but doesn't give a way to change it. It also allows other parts of the application to receive notifications of state changes in the plug-in.

The main interface of the component is the *cPluginManager*. It hides the management of the user interface dialog allowing others only to show it to the user. Plug-in management interfaces to the plug-in side allow it only a limited access to the plug-in management module.

There are three interfaces shown to the plug-ins which interface with application. These are *iGUIExtensionAccessPoint*, *iCoreExtensionAccessPoint* interfaces and through these the *iPluginAccessPoint* interface. These are shown in Figure 4.19.

The *iPluginAccessPoint* interface shown in the Figure 4.19 is inherited by all plug-in accesspoint classes. It defines state and event enumerations for plug-ins and ***void handlePluginEvent(PluginEvent)*** method that handles plug-in events. The parameter for the method is the event that happened in the plug-in.

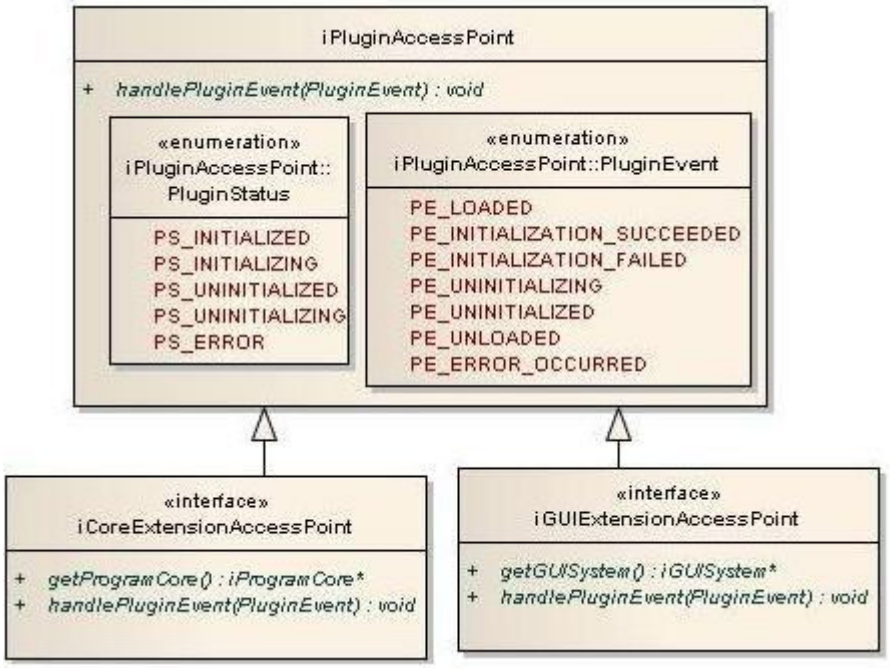

Figure 4.19. Interfaces exposed to plug-ins

The specialized interfaces *iCoreExtensionAccessPoint* and *iGUIExtensionAccessPoint*, also define their own methods that allow the plug-ins access the application functionality. The *iCorePluginHandler* has also an implementation for the ***iProgramCore* getProgramCore()*** method which allows the plug-in access to the core system functionality. The *iGUIPluginHandler* has its own variant of the function, ***iGUISystem* getGUISystem()*** method, which allows the plug-in access to the GUI system functionality.

The *PluginStatus* enumeration defines values for the plug-in's state. The states are

- PS_INITIALIZED,
- PS_INITIALIZING,
- PS_UNINITIALIZED,
- PS_UNINITIALIZING and
- PS_ERROR.

Plug-ins are required to inform the application of state changes in them. For this reason there are *PluginEvent* values that go with the *PluginStatus* values. The *PluginEvent* values are

- PE_LOADED,
- PE_INITIALIZATION_SUCCEEDED,
- PE_INITIALIZATION_FAILED,
- PE_UNINITIALIZING,
- PE_UNINITIALIZED,
- PE_UNLOADED and
- PE_ERROR_OCCURED.

5. APPLICATIONS

5.1. ITER project

The ITER project was the main reason for designing and implementing IHA3D visualization software. It is part of a distributed ITER service robot monitoring and control system. IHA3D is used to visualize the robot in the reactor, initialize haptic control and receive control data from it and communicate with various parts of the system.

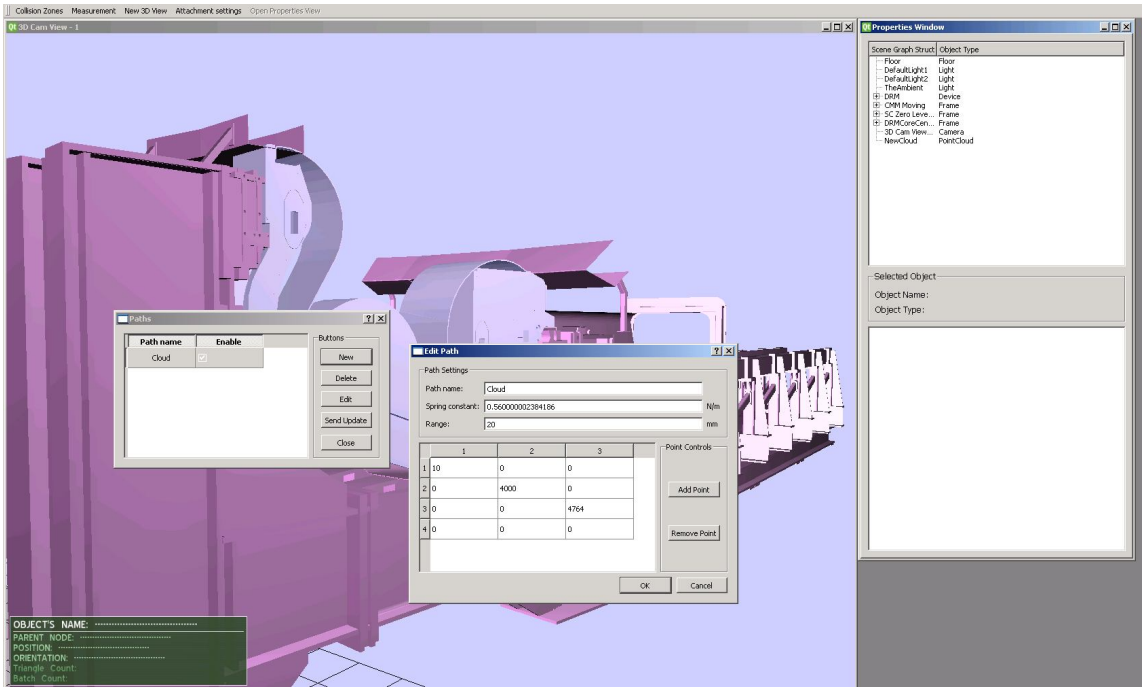

Figure 5.1. ITER application with some editing dialogs

At first, this application was meant to communicate with the other systems using UDP and TCP packets only. Later, this changed when a more advanced and reliable communication method, DDS, was taken as part of the project. The aim was to test its usefulness in the project environment and for other uses.

5.2. Fusenet project

The Fusenet application of IHA3D is intended mainly as a tool for teaching under graduate students about the ITER fusion test reactor currently being constructed in Europe. It can be used to present virtual tours of the ITER reactor and display information of the reactor and other fusion related topics. It offers an option to interactively explore the reactor if the virtual tours are not of interest.

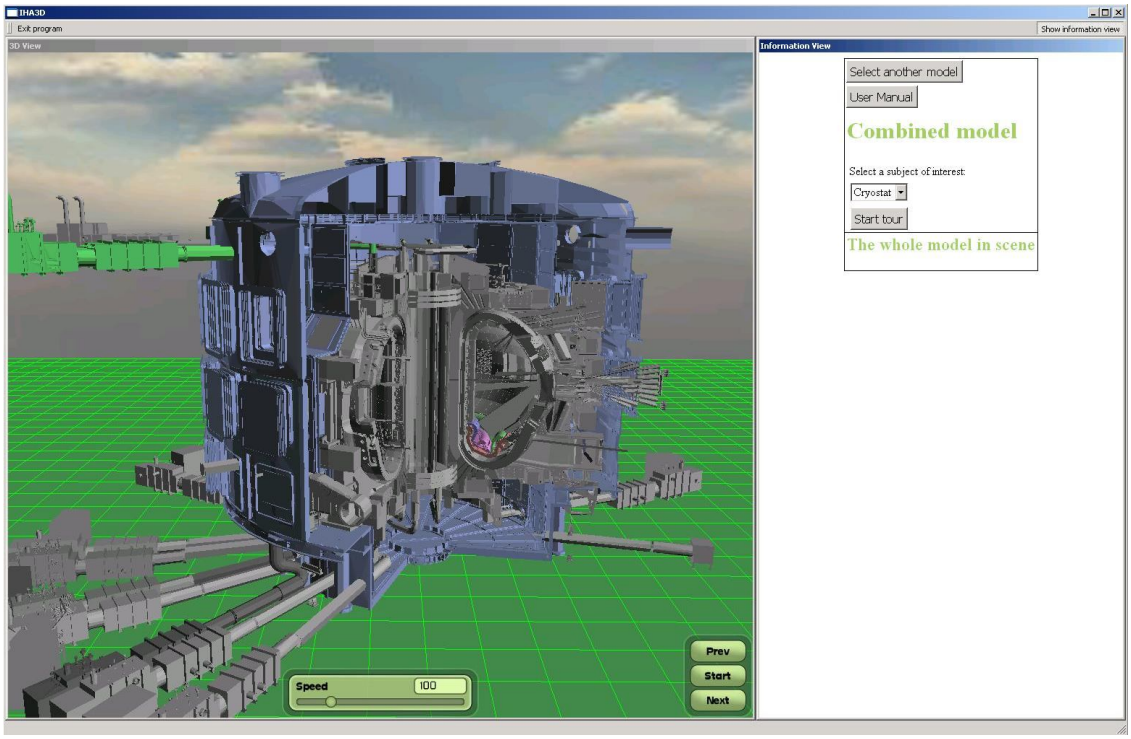

Figure 5.2. ITER virtual tour running on IHA3D

5.3. Comau visualization

The Comau application of IHA3D is intended to be used in a different project as a way to plan and visualize movement of an industry robot. This enables the testing of control systems even when the robot is not in use. It uses Ethernet connection to receive commands from the control system.

5.4. Hydraulics simulation course

This application is the most basic of them all. It is installed on a computer simply by copying and it is ready to run and receive control information.

The course application of IHA3D was designed to be used on a course visualizing hydraulics control system's output. The control system runs on a real-time LabView computer simulating a hydraulic many cylinder boom. It calculates joint values and sends them over an Ethernet connection to the computer running IHA3D. The application shows the boom orientation in the 3D scene.

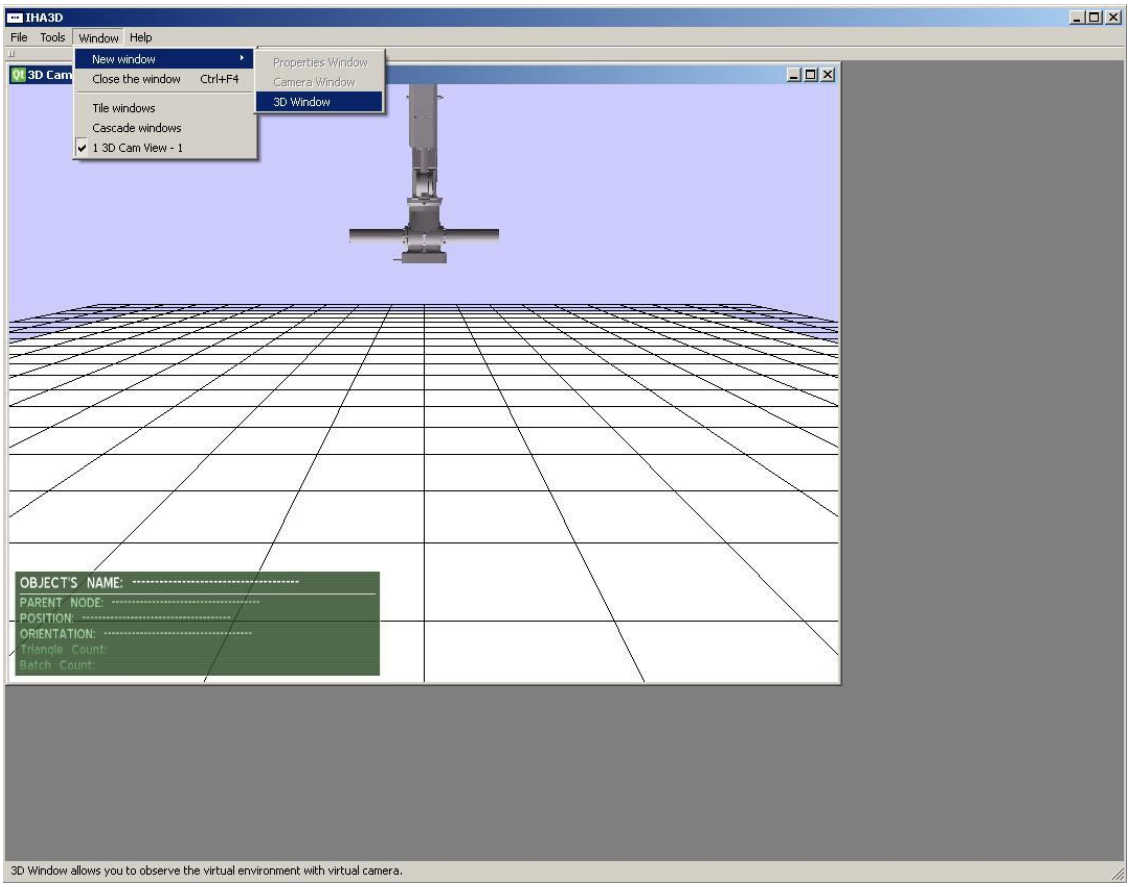

Figure 5.3. Course application with one 3D view opened

Odd about this application is that despite its apparent simple structure, it still seems to require more computational power from the computer that the Fusenet version.

6. APPLICATION EVALUATIONS

The presented evaluation results don't duplicate data associated with many of the projects. The results are mostly marked with the project they are most notably associated with. For this reason, the results may seem unfinished.

6.1. ITER project evaluation

This is the first software product of this product-line. It was used as a basis for all the other product applications thus providing the base for the product-line architecture.

- Performance:
  - Response time to human interface interaction is within 33 ms. This is fast enough for the human eye.
  - Throughput of the events should be more efficient as event buffers easily began to increase in size under slightly heavier loads on single-core machines. This is not a problem on a quad-core environment it is meant to run on, but the test revealed a bottleneck in the design.
- Modifiability: Modifying the software by changing the code is costly if the new functionality crosses threads.
- Interoperability: Once the network settings are correct, the software is a well functioning part of the whole distributed system. Especially the DDS-communication works very well for haptics control and for receiving other commands.
- Availability: The software is stable. It has no memory leaks which would limit its uptime. It was tested to run continuously for a week without problems.
- Extensibility: The software is extensible via plug-ins, but in some cases they require also something new to the existing code. This seems to indicate that the interfaces are not final as yet.

6.2. Fusenet evaluation

This software application is different from the rest as it is the only one in public distribution. It has many game-like properties like playing audio files, but it also features a web browser for information display. The information content is adaptable by the Fusenet foundation.

- Performance:
  - Response time to human interface interaction is within 33 ms. This is considered fast enough for the human eye.

  - Throughput of the events was fast in this application due to small redesigns in the visualization component. This shows that the inefficiencies can be solved with component-level redesign and evolving the product-line architecture.
  - Graphics rendering and model loading times were under a load test in this project as the size of the complete model was huge. The program was responsive even on single-core machine.
- Modifiability: This is mainly a variation of visualization plug-in. Much of the functionality is the same as in the ITER application.
- Reusability: This application required a lot more new code, some for the core but most using a plug-in. Half of the code in this project is new.
- Interoperability: The information content of this product can be help in an internet server and updated as a web page. This property improves the usability and maintainability of the product considerably.

6.3. Comau evaluation

This software application has only a little reduced functionality as the user knows what he is doing. Its reusability was high as all functionality already existed. The only thing needed was to remove unnecessary components and build the robot model. This application can be seen as a test for extending the architecture with more advanced networking.

- Performance: The DDS communication seems to be more efficiently implemented than normal networking as there are no problems with neither networked nor rendering performance.
- Interoperability: The DDS communication is used to receive joint angle information from the control system for industry robot simulation. This is working well.
- Usability: After unzipping and setting the DDS system environment variables the product is ready for use. It is only automated to load the robot model at startup to lessen repetitive work.
- Extensibility: The DDS communication capability is added via plug-in. The used DDS implementation requires system environment variables so it is not feasible to include it to the product-line functionality.

6.3.1.1 Hydraulics simulation course evaluation

This application had very little special requirements. Its reusability was high as all functionality already existed. It was required only to remove unnecessary components and to build the visual model. For availability requirement it is enough for the program

to run for a day without crashing. Curiously, this was the application that seemed to require computing power the most.

- Performance: Through put of the events should be more efficient as event buffers easily began to increase in size when events were received via network faster than 20 ms/msg on dual-core machine.
- Reusability: Reuse of existing code was high as only minor fine-tuning was needed after the model was built.
- Interoperability: Ethernet network is used to relay joint values for the boom every 20 ms.
- Usability: After unzipping the product is ready for use. It is highly automated so students don’t need to do any work unrelated to the project work. The user interface is simplified and only core functionality is left.

7. PRODUCT-LINE EVALUATION

This architecture was not originally supposed to be a product-line even if it did have some properties supporting it. This introduced its own challenges. The API and other supporting libraries were designed from the start to help design and implement new extensions for the system. Later, there became the need to implement slightly different applications of the software, and this lead to the unnoticeable drift to product-line architecture. These applications are used as scenarios to evaluate the architecture. They imply that the existing functionalities should be refactored to evolve more coherent product-line architecture.

7.1. Modifiability

Adding new functionality that crosses threads seems to be costly. This could be improved by refactoring some of the existing plug-in functionality into the product-line and possibly moving them to the main thread. Some of the important functionality has been placed in two main plug-ins when it either should be in a more specialized plug-in or part of the core executable.

The rendering engine could be included to the product-line as a component library with slightly evolved interface. This would also improve the efficiency of the different applications, increase modifiability and would still allow the rendering engine to be changed if necessary.

7.2. Usability

The common libraries for the API usage are at the moment in one file. This is restrictive as it forces the extensions to include a lot of unnecessary functionality that may in some cases limit them by introducing unwanted dependencies. The common libraries should be broken into smaller libraries for more targeted usage.

The large plug-ins also add to the programmer’s work load as some changes have to be made in 2 different components. This can be fixed by incorporating the scenegraph to the executable and centralizing the functionality by integrating it to the PLA.

7.3. Reusability

The software has an API and helpful libraries to make it easier to develop different extensions. There are also skeleton projects containing what is needed to compile a working plug-in to get the development started quickly. Rest of the functionality needs only to be added within a plug-in to get a usable application.

7.4. Extensibility

Much of the functionality can be extended. Some of the changes may require modifications and additions to the existing codebase, but they have not so far been a problem. The event handling extends as needed. The core system does not need to know what type of events it relays as the components handle it.

8. CONCLUSIONS

The main point of this thesis was on the quality requirements which were mainly satisfied. Of course, there is always room for improvement.

When designing the application, the performance requirement based on the assumption that there are always at least 4 processor cores available in the hardware is a bad one. The application needs to run efficiently even on older, less powerful, computers as was shown in Fusenet project. During that project, the application was tested on a very slow, single-core, computer. The test showed IHA3D to be flexible and efficient enough to be usable even on older computers. The problem on slow computers is that they have a very limited number of updates, events, the system can handle before becoming unusable. This limit is around 90 events per second as changing the thread context is a slow process. This is efficient enough, but it can be improved by refactoring the software in the future.

The availability requirement was tested while the ITER application was running constantly for a week in the test environment of a quad-core PC. The long running test didn't reveal any noticeable memory leaks that could endanger the applications stability. One encountered problem seemed to be in repeated starting and stopping of DDS-communication plug-in. This also required the DDS core to be restarted occasionally, but it is not a major bug.

While developing the different applications of the PLA, the modifiability and extensibility requirements were tested extensively. All the applications are somehow different from each other and have brought something new to the design. The extensions seem to integrate as part of the program without too much difficulty once a stable framework has been achieved. In some cases the modifications took a lot more effort than in others. This implies the need to refactor parts of the design in the future.

The reusability requirement was considered when developing the architecture. Many of the components and extensions share the same kind of structures and code sections with no or minor changes. This was further improved by creating an API and a library file for developers to make use of and hopefully shorten the development time.

The application can be considered to be interoperable. It implements TCP/UDP network management component that can be extended with protocols based on these. The communication capabilities of the application were further improved in the ITER project by implementing a DDS-communication plug-in for more advanced data transfer protocol.

Usability of the new application could be improved in the future. There are some features the existing IHA3D application has that are still missing in this newer application. On the other hand, this newer application is easier to develop as it is

implemented using third party frameworks and libraries that will be in active development for many years to come. From the user's point of view, some features could be automated while others should be improved in the future.

All in all, the architecture seems to work. The applications do what they are meant to do and the architecture enables them to be further developed and improved as more features are required.

REFERENCES

[1] Koskimies, K., Mikkonen, T. Ohjelmistoarkkitehtuurit. Helsinki, Talentum 2005.

[2] Implicit Sharing [WWW] [accessed on 24.10.2013]. Available at: http://qt-project.org/doc/qt-4.8/implicit-sharing.html

[3] Signals & Slots [WWW] [accessed on 24.10.2013]. Available at: http://doc-snapshot.qt-project.org/4.8/signalsandslots.html

[4] The Event System [WWW] [accessed on 24.10.2013]. Available at: http://qt-project.org/doc/qt-4.8/eventsandfilters.html

[5] Software Design Pattern [WWW] [accessed on 24.10.2013]. Available at: http://en.wikipedia.org/wiki/Software_design_pattern

[6] Observer pattern [WWW] [accessed on 24.10.2013]. Available at: http://en.wikipedia.org/wiki/Observer_pattern

[7] Using Qt and Symbian together (Figure 6: PIMPL Class Overview) [WWW] [accessed on 24.10.2013]. Available at: http://www.developer.nokia.com/Community/Wiki/Using_ Qt_and_Symbian_C%2B%2B_Together

[8] Event driven architecture [WWW] [accessed on 24.10.2013]. Available at: http://en.wikipedia.org/wiki/Event-driven_architecture

[9] Model/View Programming [WWW] [accessed on 24.10.2013]. Available at: http://qt-project.org/doc/qt-4.8/model-view-programming.html

[10] Reusability [WWW] [accessed on 24.10.2013]. Available at: https://en.wikipedia.org/wiki/Reusability

[11] Usability [WWW] [accessed on 24.10.2013]. Available at: https://en.wikipedia.org/wiki/Usability

[12] Interoperability [WWW] [accessed on 24.10.2013]. Available at: https://en.wikipedia.org/wiki/Interoperability,

[13] Availability [WWW] [accessed on 24.10.2013]. Available at: https://en.wikipedia.org/wiki/Availability

[14] Extensibility [WWW] [accessed on 24.10.2013]. Available at: https://en.wikipedia.org/wiki/Extensibility

[15] Opaque pointer [WWW] [accessed on 24.10.2013]. Available at: http://en.wikipedia.org/wiki/Opaque_pointer

[16] Qt [WWW] [accessed on 16.02.2014], Available at: http://qt-project.org/resources/getting_started

[17] Dispatcher pattern [WWW] [accessed on 16.02.2014], Available at: http://msdn.microsoft.com/en-us/magazine/cc301357.aspx

[18] Performance [WWW] [accessed on 16.02.2014], Available at:

http://msdn.microsoft.com/en-us/library/ee658094.aspx

[19] Modifiability [WWW] [accessed on 16.02.2014], Available at: http://sa.inceptum.eu/tool/modifiability

APPENDIX 1 API DEFINITIONS

This chapter describes the functionality implemented for common use and the interfaces needed in most projects. The application provides an API to help plug-in developers in their task.

A.1. Data types

API interfaces include some implementations of data types to help in graphics computation and passing data over the API boundary. Math data types implemented are quaternion, 3D vector and 3x3 matrix. The math types can't pass over the API boundary. The API includes implementations of events, requests and values to pass the information between components.

A.1.1. Values

Values transmitted with events are stored and conveyed by objects implementing iValue and iValueGroup interfaces. The implementations of these interfaces should be included in every component using events. The Figure 8.1 shows the class structure of value classes and interfaces in the API.

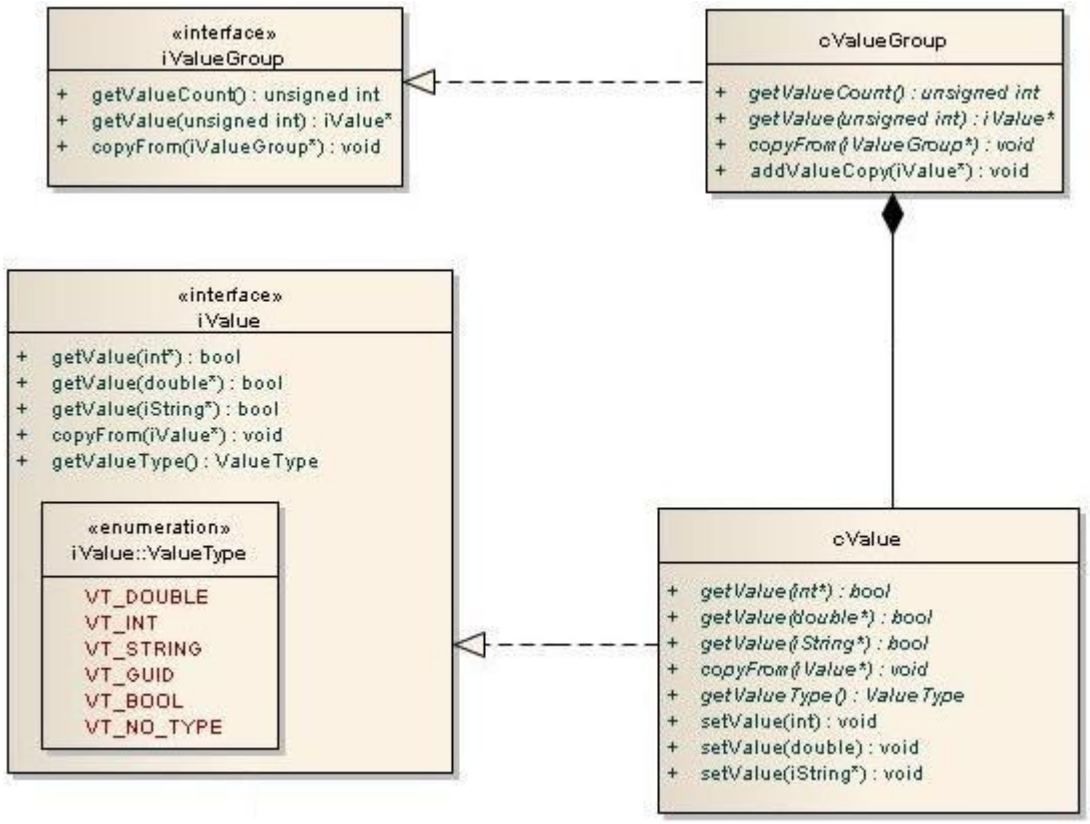

Figure 8.1. Value class structure

Value and valueGroup both have an interface class and an implementation class. The interface classes are meant to be accessed beyond API boundaries. For this reason, they show only functionality that is needed to access the data beyond an API boundary. Implementation classes define an object's data and implement the methods for changing an object's data content. The value and valueGroup classes offer the option of copying the data from another object and this way bypassing the restrictions of API boundaries. A *cValue* object stores one value and its type. The type is enumerated `ValueType` type defined in the *iValue* interface. A *cValueGroup* object can store an arbitrary number of *cValue*s.

A.2. Communication data types

Events store their parameters in the cMessage class type objects. It carries the data in its cValueGroup type member variable. This relationship is shown in both **Error! Reference source not found.** and Figure 8.2.

A.2.1. Message

As the **Error! Reference source not found.** shows, a message consists of iMessage interface and its implementation cMessage. The iMessage defines an interface that can cross the API boundary and is used to access and copy the data. The cMessage implements the functionality to change the message's data content. Message contains one cValueGroup type member variable to store the message parameters. Message contains variables identifying it uniquely and identifying its type.

A.2.4. Events

Events are created and sent as *cEventGroup*s. Events are a way to inform all interested listeners that either the program state or model state has changed. Events are sent to the core using the *iCommunicationHandler* interface.

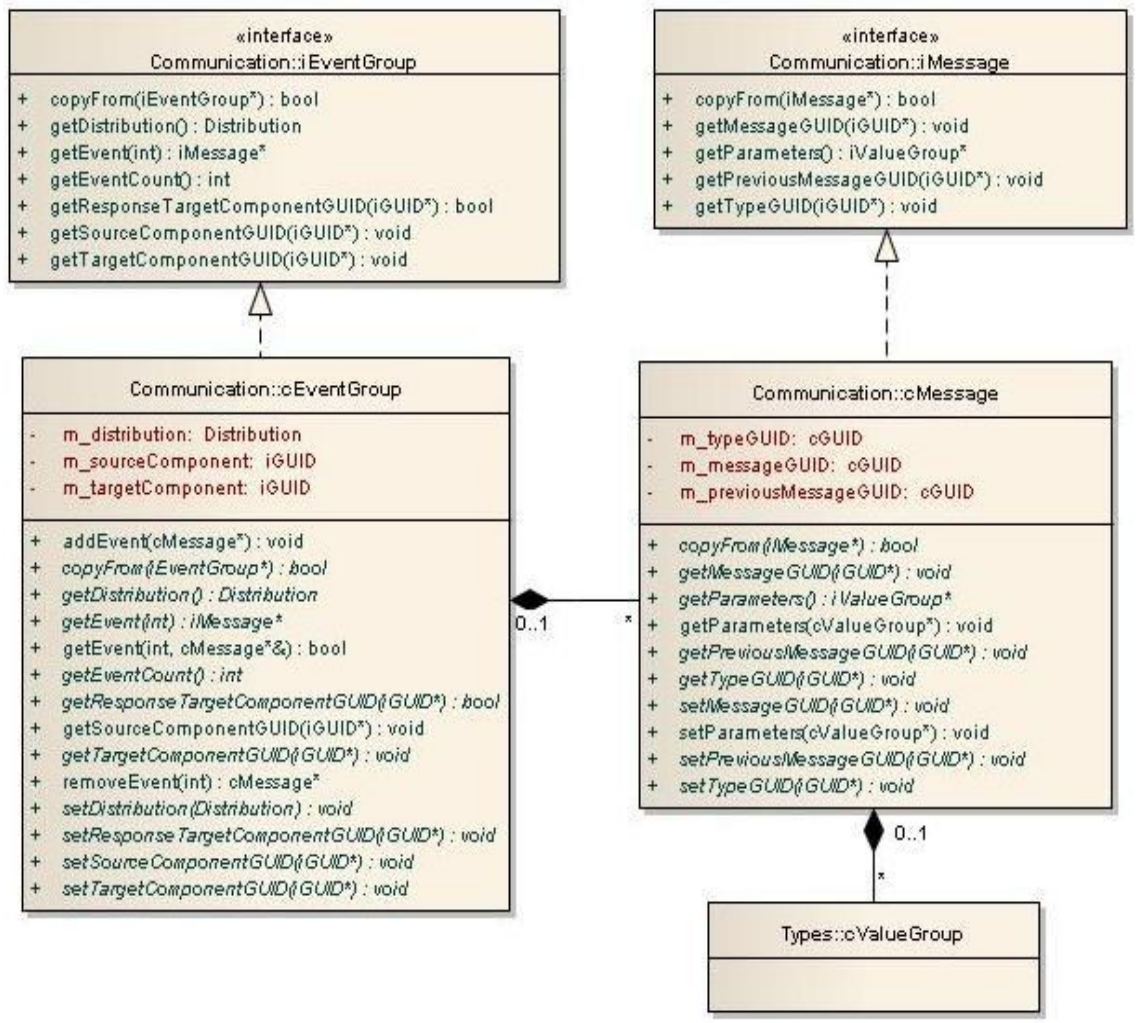

Figure 8.2. Event class structure

The *cEventGroup* implements the *iRequestGroup* interface to iterate through the data it contains. In addition it provides methods to add new data to the request. The object can contain several *cMessage* objects each one acting as a single event in the group. The instances of *cMessage* object act as value storage for the event's parameters.

A.2.5. Event definition

Here is an example definition of an event:

```
/*!   \brief      Event informing of a created object.
      \param      iBasicEntity::identifier      id of object.
      \param      GUID        The GUID of the object (if any).
      \param      iBasicEntity::identifier      The   id   of   parent
object.
      \param      GUID  The GUID of the object type.
      \param      cString     The name of the object.
      \param      cString     The description of the object.
*/
const cGUID OBJECT_CREATED( 0x8d07271b, 0x489a, 0x4d3a, 0x93, 0xd6, 0x49,
0x86, 0xb6, 0xa1, 0xce, 0xf );
#define EVENT_VERSION_OBJECT_CREATED 1
```

When a plug-in brings custom events to the system, they are defined in a header file that is included in every component that uses the new events. There may also be some configuration file defining events that all interested plug-ins can load and parse. This is left to the developer to decide.

The events must have a GUID identifying them uniquely when sent as a reply to a request in the system. It also defines a version number for the event. This is useful to recognize incompatibilities in plug-in development.

An event may have a number of values attached to it carrying the necessary data for event handling. The event system restricts the value types used in the events to those enumerated in iValue interface. They are the most often used data types so as to avoid causing problems during plug-in development.