The Use of Object-Oriented Techniques and CORBA in Astronomical Instrumentation Control Systems.

N.A.Dipper and C.Blackburn
Centre for Advanced Instrumentation, Department of Physics, University of Durham, Science Laboratories, South Road, Durham, UK, DH1 3LE

ABSTRACT

Control software for astronomy matches the ever increasing complexity of new large instrumentation projects. In order to speed the development cycle, object-oriented techniques have been used to generate loosely coupled software objects and larger scale components that can be reused in future projects. Such object-oriented systems provide for short development cycles which can respond to changing requirements and allow for extension. The Unified Modeling Language (UML) has been used for the analysis, design and implementation of this software. A distributed system is supported by the use of an object broker such as CORBA. These techniques are being applied to the development of an instrument control system for the UK spectrograph within FMOS (Fiber-fed Multi-Object Spectrograph). This is a second generation instrument for the Subaru Telescope of the National Astronomical Observatory of Japan.

Keywords: object-oriented, Python, CORBA, UML, design patterns, control systems, FMOS

1. INTRODUCTION

Astronomical instrumentation is becoming ever more complex. This complexity is reflected in the instrument control system (ICS) software. A large amount of effort is expended on these systems with an associated high cost. The application of object-oriented coding methods and reusable elements of both design and actual code can contribute significantly to a reduction in the development time and therefore the cost of control systems. In section 2 of this paper, we describe the general principles behind the approach being taken to such ICS development at the Centre for Advanced Instrumentation at Durham. This includes a discussion of object-oriented techniques, design patterns, the use of object brokers such as CORBA to provide middleware, and system design using the Universal Modelling Language (UML). In section 3 we present an example of the use of these techniques as applied to the control software for the UK spectrograph for the FMOS instrument that is being developed for the Subaru telescope at the NAOJ on Hawaii. The FMOS UK ICS is being developed using the Python language with CORBA middleware. The practical problems of interacting with non-object-oriented components of the system are also addressed.

2. REUSABLE OBJECT-ORIENTED CONTROL SYSTEMS

Astronomical instrument control systems in the past have been specific to a single instrument and made relatively little use of either design or code from previous systems. We now wish to work towards developing a set of reusable software solutions within the astronomical community. An important principle in all modern software is the speed of the development cycle. To achieve this aim, it is necessary to design and build systems that draw on existing expertise and add to the pool of reusable components. In looking to develop such systems, certain general principles must be considered.

2.1. Object-oriented Coding

Object-oriented software provides many features which facilitate reuse. Objects can be designed to encapsulate their data and implementation. Objects should be designed to be highly coherent, containing the necessary functionality for their purpose, no more, no less. Such objects can be loosely coupled thus reducing dependencies within the system. This allows new objects to be used in existing systems, existing objects to be reused in other systems and objects to be modified, all with less effort than in highly coupled systems. The eventual target is to design a homogenous system of object-oriented code. However, control systems for large astronomical instruments are often developed within a large international collaboration and code standards will vary. Where necessary, proxy objects can be used to interface to non-object-oriented systems.

At a more coarse-grained level several closely related objects may be bound together to create a component. With a well-defined interface and documented functionality a component provides a higher level of reuse. Some languages offer direct support for component-based software while in others software conventions, such as modules, can be used to support the component idiom.

Several pure and hybrid object-oriented languages have now become popular, such as C++, Python and Java. The example described in section 3 makes use of the Python language.

2.2. Middleware

Objects within separate, sometimes distributed, subsystems can use a suitable middleware, such as CORBA, to facilitate inter-object communication. Messages and data can thus be sent from one object to another object regardless of their physical location within a system. Suitable middleware also allows systems written in different languages and running on different platforms to communicate easily.

Many control systems used in astronomical instrumentation could be categorised as heterogeneous, distributed systems. To facilitate communication between the elements that comprise such a system a specific middleware must be chosen. There are now many stable products available and there is a wide source of information and expertise at hand. Systems with differing requirements may benefit from different middleware solutions. CORBA has been selected in the example described in section 3.

2.2.1. CORBA

CORBA is a mature product, it has been around for over a decade. As such, it is the middleware that has been most commonly adopted in astronomical applications. It is available for a wide variety of platforms such as Linux, UNIX variants and Microsoft Windows. Communication between different systems involves no extra work on the part of the application programmer as all communications are handled by the CORBA system. CORBA supports a wide range of languages. There are language bindings available for C++, Python and Java, among others, utilised through a language-independent interface definition language.

2.2.2. SOAP and XML-RPC

SOAP and XML-RPC have wide platform support and are an open standards, but they are not object-oriented. They both make use of socket connections through the HTTP protocol. They have no specific language support so language bindings must be written or obtained elsewhere. They are not as efficient as CORBA as they communicate with relatively large text-based XML packets.

2.2.3. ICE

Internet Communications Engine (ICE) is a relatively new product that promises to be a simple, fast, multi-platform, object-oriented middleware. Like CORBA it uses an interface definition language to bind to multiple languages. However, at present it only offers support for the C++ and Java languages.

2.2.4. Pyro

Python Remote Objects (Pyro) is simple, fast and multi-platform but it is tightly bound to the Python language and thus offers no direct support for other languages. At CfAI, Pyro is used for laboratory control systems during the design phase of new instrumentation. It allows us to build a library of Python objects, mainly for hardware control, on a distributed system with multiple platforms.

2.2.5. Others

J2EE and RMI are Java specific; DCOM, COM & COM+ are platform specific, and not easily combined with open technologies. Finally, RPC and sockets require the application programmer to develop the lower level communication interfaces, ie to create their own middleware.

2.3. Design Patterns

Design patterns allow expertise to be reused by drawing on documented solutions to commonly occurring problems. Design patterns do not present a reusable coded solution but rather a recipe for a solution which must be implemented appropriately for a given application. For instance, the observer pattern[1] can be used when some objects depend on state change in another object. An example of the use of the observer pattern is given in section 3.3.

2.4. Modeling Languages - UML

A diagrammatic modelling language should be used to assist in the correct development of a software system and in documenting that system. Systematically designed and clearly documented software improve the reusability of design. A modelling language should have a number of techniques available for modelling functional, structural and behavioural aspects of systems. Such techniques should allow the model to be investigated at different levels of granularity. The Unified Modeling Language expresses the analysis, design and implementation of the software using a number of graphical and textual techniques.

2.5. Graphical User Interfaces

Graphical user interfaces are an important part of modern software systems. Even when control of a system is ordinarily achieved through a script of commands, a GUI can provide vital visual feedback and engineering access for testing and configuration. A suitable GUI system should provide access to a range of widgets allowing the creation of complex interfaces in the language of choice.

3. FMOS: AN EXAMPLE SYSTEM

The Astronomical Instrumentation Group (AIG) at the Department of Physics is a part of the Centre for Advanced Instrumentation (CfAI) of the University of Durham. CfAI has a work package to develop the Instrument Control System (ICS) for the UK infra-red spectrograph that forms a part of the Japanese/UK second generation instrument Fibre-fed Multi Object Spectrograph (FMOS) for the Subaru telescope at the NAOJ on Hawaii[2]. Typically for a large astronomical instrument, FMOS is being designed and constructed by an international collaboration in Japan, the UK and Australia. FMOS consists of two similar near infra-red spectrographs mounted above one of the Nasmyth foci of the Subaru telescope. One of these spectrographs is being built at the University of Kyoto in Japan. The other is being built by a UK collaboration consisting of the University of Oxford, the Rutherford Appleton Laboratory and the CfAI in Durham. Infra-red light is fed to the spectrographs via about 50m of fibre optic cable from the focal plane positioner. This novel positioner ECHIDNA[3,4] allows the simultaneous observation of 400 different astronomical objects, and has been developed by the Anglo Australian Observatory in Australia.

The full ICS to control this complex instrument is also being developed by different parts of the same collaboration. Parts of the system are therefore not object-oriented and use different languages and middleware that must be accommodated. The design and implementation of the UK part of this ICS is used here as an example of a purely object-oriented system with code developed with reuse for future projects in mind. The solutions developed for this system are used to illustrate the general principles that have been outlined in section 2. The software is object-oriented, written in Python and uses CORBA for middleware. The FMOS UK ICS, along with its various communication paths, is shown in Figure 1.

3.1. Object-oriented Coding

In the choice of a language for this development, the most important aspects are speed of development and the ability to generate reusable design and code. There are a large number of object-oriented languages now available. Of these, C++, Python and Java are the most popular. The use of a scripting language increases the speed of software development. To this end, and unusually in an astronomical ICS, we used a scripting language Python, not only for the Graphical User Interfaces (GUIs) but also for the core of the ICS. Python is a highly object-oriented language which allows software to

be developed using classes which are highly cohesive and loosely coupled to other classes. In addition, Python is a very readable language and imposes a clear structure on the code. As a scripting language it offers many facilities for binding together code written in other languages, wrapping existing code for reuse and utilising operating system services. It is byte compiled and thus very fast for an interpreted language.

Python offers a modular source code structure. Modules (files) can contain several classes and additional code to create a component. The scope rules and module import facility support the use of components to create a larger application. A component level view of the ICS is shown in Figure 2.

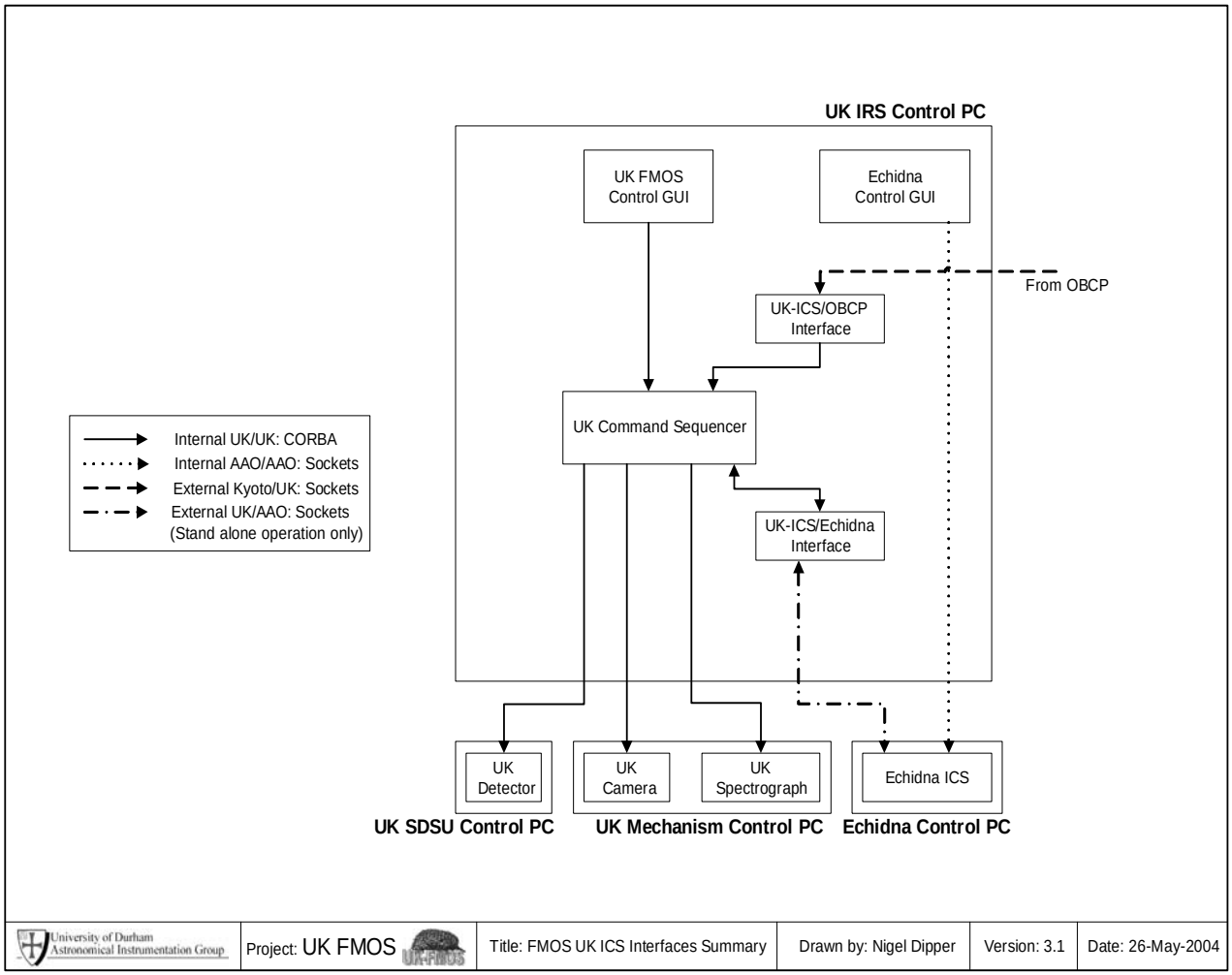

Figure 1: The Elements of the FMOS UK ICS and their Communications Protocols

3.2. Middleware

3.2.1. CORBA

The code for the FMOS UK ICS is distributed both between several computers and between many applications within those computers. Middleware can be chosen that performs all of the communication that is required between these various elements. Many different middleware solutions are now available as summarized in section 2.2. CORBA, Common Object Request Broker Architecture, was chosen for this purpose. Other middleware systems were considered and ruled out for a number of reasons. CORBA is truly object-oriented. This means that not only are the bindings between it and object-oriented languages natural, the design paradigm is consistent between the application and the middleware. It is available for a wide variety of platforms including Linux, which is being used throughout the FMOS project, and supports the Python language that has been selected. CORBA uses a language-independent Interface Definition Language (IDL) to specify the attributes and methods of objects. This can then be compiled to produce language-specific stubs.

Some middleware products have a large overhead and can slow down inter-process communications substantially. CORBA is efficient. It marshals messages and arguments and then sends compact binary packets. CORBA systems are also scalable and can handle any number of objects in a distributed system. This allows projects to scale easily as requirements change.

CORBA is an open standard. This has lead to a number of competing products being developed, some commercial and some freely available, but all are inter-operable with each other. We have chosen to use the OmniORB implementation of CORBA, developed originally by AT&T. This is one of the most up to date and fully standards compliant ORBs, as defined by the Object Management Group (OMG). It meets all of the requirements for the FMOS UK ICS, including a binding via IDL to the Python language.

3.2.2. Other Middleware

In an ICS that is being developed by an international collaboration, the adoption of identical protocols and standards by all parts of the collaboration in order to generate homogeneous object-oriented code is an ideal goal that cannot always be achieved. More typically, code standards will vary and object-oriented techniques are not yet universally adopted. Equally, the ICS for a new instrument must interface to existing legacy systems and protocols within the overall telescope control software. This is the case with the FMOS UK ICS. Figure 1 includes two examples of interfaces between the CORBA systems and other communications protocols. These interfaces allow the UK ICS to communicate with the Japanese overall FMOS control system known as OBCP and with the Australian focal plane positioner, Echidna. Both of these systems provide a low level interface based on the direct use of sockets. Our solution to this problem is an interface between the two protocols taking the form of proxy objects. These can be addressed as normal objects within the CORBA system. From the point of view of the OBCP or Echidna, they look like any other sockets client. These proxy objects could very easily be converted to actual objects when or if the lower level interfaces are replaced by CORBA within these systems. The functionality of the objects would remain unchanged and require no changes within the FMOS UK ICS. These proxy objects are also shown in the UML component diagram in Figure 2. They could be implemented on either the client or server machines with no significant changes to the ICS.

3.3. Design Patterns

Many of the requirements for an astronomical ICS are common to both other instruments and other non-astronomical systems. Often, a well known design solution exists that provides the required functionality. Many of these design patterns have been published[1]. An example within the FMOS UK ICS is our use of the Observer pattern.

There are many situations within an ICS where the status of a sub-system will change at an unknown time. The monitoring system can typically poll the sub-system until its status changes. This is very wasteful of both CPU and network bandwidth. The observer pattern consists of objects acting as observers on the status of another object. They thus register an interest in the state of that object. Once the observed object changes its status, it will inform all other objects that are observers on it that its status has changed. They can each then undertake whatever action is required.

This pattern is used within the FMOS UK ICS to inform higher level objects when the status of a lower level object changes. In this way, the display of an element of the system shown in a GUI can be updated automatically, without polling, whenever the status of the relevant object changes for any reason. This process is illustrated in the sequence diagram shown in Figure 4.

3.4. Modeling Languages - UML

The Universal Modeling Language, UML, has become the de-facto standard in the graphical representation of the design and interaction of software objects. Diagrams for the design of the FMOS UK ICS were generated using a case tool. Many UML case tools are now available. The most commonly used is Rational Rose. However, we have selected the Object Domain tool. It has most of the functionality of Rose with the added benefit of the ability to generate stub files in the Python language. Thus, once the graphical UML design of an application is complete, the case tool generates the overall structure of the code, including class definitions, inheritance etc. The required functionality can then easily be inserted into these files. The case tool also generates the bulk of the documentation for each application.

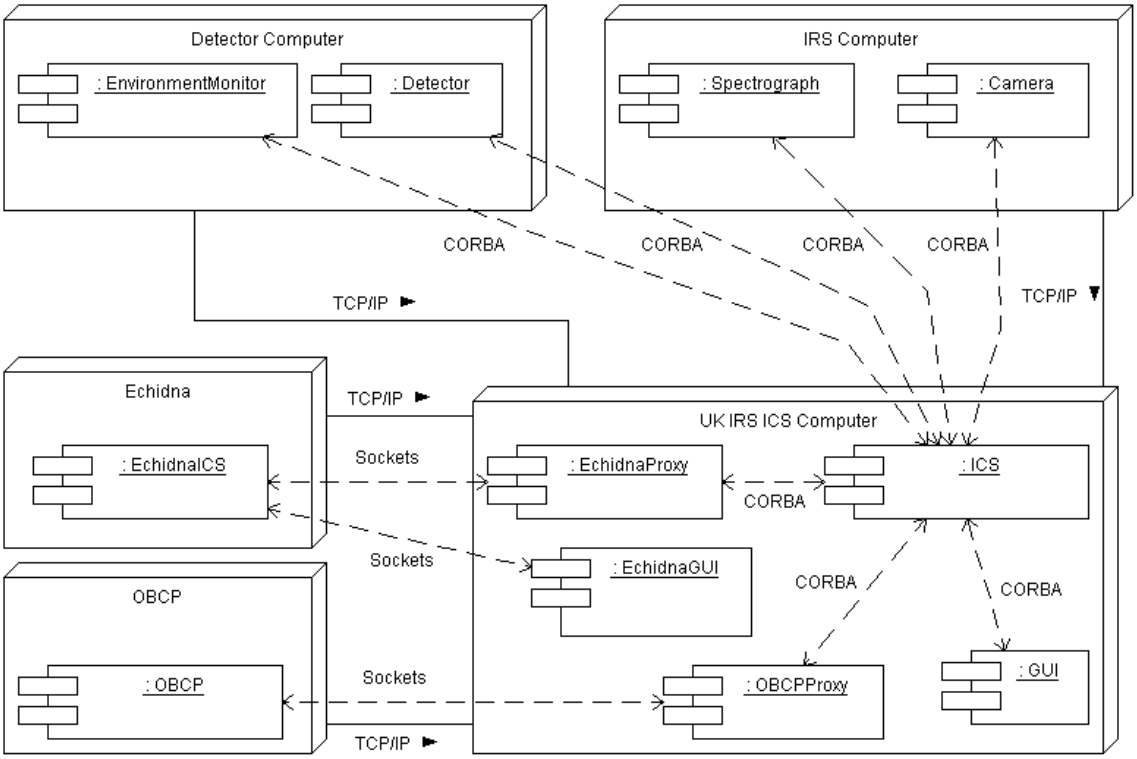

Figure 2: Deployment diagram for the FMOS UK ICS and relevant external elements

3.4.1. Example UML Diagrams

Figures 2 to 4 provide examples of the UML diagrams used in the design of the FMOS UK ICS. Figure 2 shows a top-level deployment diagram. This illustrates some of the components internal to the FMOS UK ICS and the external FMOS components with which the UK system must communicate. The diagram also shows the machines on which the components will be instantiated and the communications protocol connecting those machines. Finally, the communication dependencies between components are shown.

Figure 3 shows the structural detail of one component with the FMOS UK ICS, the spectrograph mechanism control system. The component comprises a number of classes and the associations between those classes. The interface of the single controlling Spectrograph class is the provided interface for this component, ie it is available to other components

in the system. The MotionCard class communicates with the Galil motor control card and is the only hardware dependent class in this component.

Figure 4 illustrates the effects of a system operation on the Spectrograph object. It shows the time ordering of messages between the objects when a message requesting an element be moved is received. After the MotionCard object has asked the Galil card to move the motor a change of state is propagated to dependent objects, via the changed() and update() messages, using the Observer pattern.

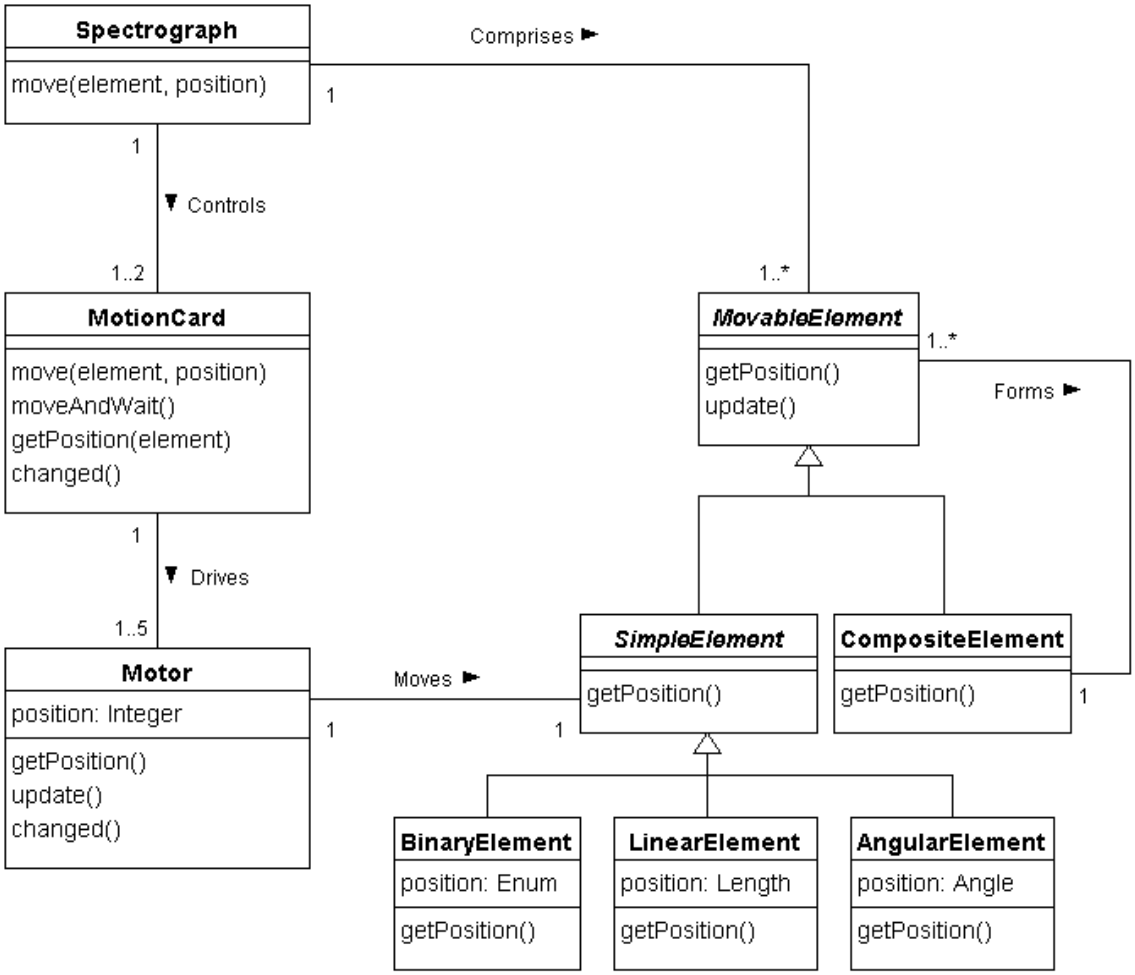

Figure 3: Class diagram for the Spectrograph Mechanism Control System

3.5. Graphical User Interfaces

The Tkinter GUI system is used in the FMOS UK ICS. Tkinter makes use of the existing and widely implemented Tcl/Tk GUI system. It is implemented in Python and provides a basic set of widgets that can be used to build more complex composite widgets. A large number of third party reusable widgets is available and many, such as Python Mega Widgets (Pmw) have been used in the FMOS UK ICS control GUI.

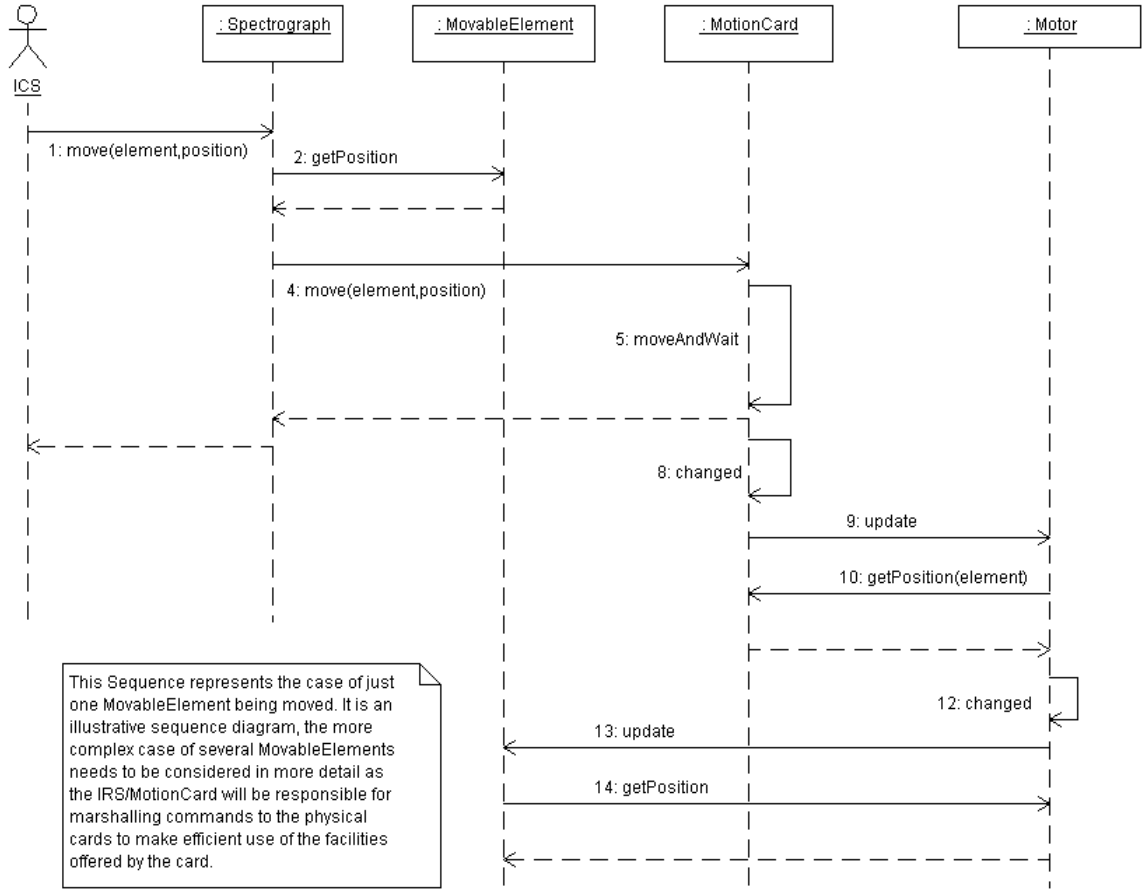

Figure 4: Sequence diagram for a system operation of the Spectrograph Motion Control System

4. CONCLUSIONS

Modern astronomical instrument control software is complex. Some aspects of this complexity can be reduced by using modern software techniques that facilitate reuse, and by using existing technologies in both software and design.

In this project object-oriented design methods have been applied. This design has been facilitated through the Unified Modeling Language. Existing documented design patterns have been used when commonly found problems have arisen. This has led to a series of highly coherent components that are not highly dependent upon each other. The components themselves comprise a number of objects adhering to the same principles. The software has been implemented in a readable object-oriented language that fully supports the design philosophy. A sophisticated middleware, CORBA, has been employed to broker communication between objects and components spread across a distributed system.

Reusing existing technologies and expertise has led to the complexity of the current project being reduced. Following sound object-oriented and component-based design and programming principles to create reusable components the complexity of future projects in astronomical instrument control systems will also be reduced.

REFERENCES

1. E. Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, 1995.
2. Masahiko Kimura et al, *Fibre-Multi-Object Spectrograph (FMOS) for Subaru Telescope*, Instrument Design and Performance for Optical/Infrared Ground-based Telescopes; Masanori Iye, and Alan F. Moorwood; Eds., Proc. SPIE **4841**, pp. 974-984, 2003.
3. P. Gillingham, S. Miziarsky, M. Akiyama and V. Klocke, *Echidna – a Multi-Fiber Positioner for the Subaru Telescope*, Optical and IR instrumentation and detectors, M. Iye, A. Moorwood, ed., Proc. SPIE **4008**, pp. 1395-1403, 2000.
4. P. Gillingham et al, *The Fiber Multi-Object Spectrograph (FMOS) Project: the Anglo-Australian Observatory role*, Optical and IR instrumentation and detectors, M. Iye, A. Moorwood, ed., Proc. SPIE **4841**, pp. 985-996, 2003.