An Architectural Style for Multiple Real-Time Data Feeds

Neil Roodyn
Cognitech Ltd
City Cloisters, 188-194 Old Street
London EC1V 9FR, UK
neil@cognitech.co.uk

Wolfgang Emmerich
Dept. of Computer Science
University College London
London WC1E 6BT,UK
w.emmerich@cs.ucl.ac.uk

ABSTRACT
We present an architectural style for the integration of multiple real-time data feeds on Windows-NT platforms. We motivate the development of this style by highlighting different application areas in which the style has been deployed. We present the requirements that will be met by the architectural style and discuss the design of customizable components that implement the style based on Microsoft's Component Object Model.

Keywords
Software Architectures, Architectural Style, Component-based Development

1 INTRODUCTION

Many systems in the financial industry use *data feeds* for the exchange of information. A data feed can be considered as a continuous stream of data. Data feeds are also used in the transport sector, for example in air traffic control and vehicle tracking systems. Data feeds are produced by one system and processed, filtered, viewed and archived by other systems. An example is the integration of different trading systems for financial products, which feed data to back-office systems where these trade data are processed and subsequent financial transactions are started. There are commercial providers of data feeds, such as Reuters and Bloomberg, which provide subscribers with up-to-date price information about trades that have recently been completed at the stock exchanges.

Following multiple data feeds is too labourious for humans and they are often overwhelmed by the sheer amount of information that is presented to them. The data feeds of Reuters and Bloomberg are good examples. They provide new data items every few seconds, whenever a trade has been completed at the stock exchange. It is impossible for humans to follow several of these feeds over prolongued periods of time. We refer to this situation as *information overload*. Hence, the need arises to automate the integration of data feeds so that users can follow only a single stream of data. That stream is also often filtered such that only those subsets of data are output that a user is interested in. Again, this need for data integration and filtering is generic and applies in many settings. Figure 1 contrasts human and automated feed integration.

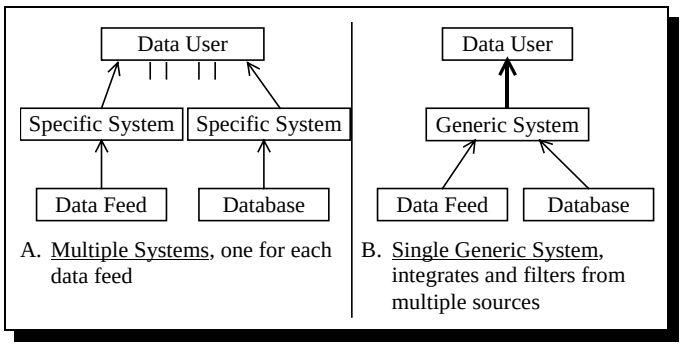

Figure 1: Multiple vs Single Feed Processor

Many data feeds have to be processed as quickly as possible. Traders at the stock exchange, for example, might miss opportunities if they are not informed about changes in the market as they happen. Hence, many of the information systems used in this setting have real-time response-time requirements. In safety-critical systems, such as cruise control in aircraft or reactor controls in nuclear power plants, real-time constraints are *hard* and could lead to desasters if the system does not respond in time. The response-time requirements in financial systems are *soft* in that slow responses do not render the system incorrect, but they would lead to a low acceptance of the system.

The platforms used in financial organizations have changed over the last two years. While they previously employed mainframes, which were then downsized to UNIX workstations and servers, institutions now start to deploy Microsoft operating systems on both servers and workstations, Microsoft's SQL Server for database applications and other Microsoft standard packages. This has resulted in a desire to be able to also integrate multiple data feeds in soft real-time on these platforms.

The main contribution of this paper is the discussion of an architectural style that can be deployed for the integration of multiple data feeds in soft real-time on Windows-NT. We refer to this architectural style as *RTD*, which stands for Real Time Data. RTD is customizable to match different demands for the integration of data feeds. We have licensed the architectural style to several different technology providers, who have instantiated RTD for the processing of data feeds in various application domains.

This paper is further structured as follows. Section 2 relates the architectural style suggested in this paper to the literature. In Section 3, we discuss the non-functional requirements that drove the development of this architectural style. We then discuss the architectural style for real-time data feed processing in Section 4. In Section 5, we describe a particular example that instantiates the RTD style. We present an evaluation of the style on the basis of several projects in Section 6 and summarize the work in Section 7.

2 RELATED WORK

Our work is largely related to research on architectural styles [9]. The literature has identified many styles, such as tuple-spaces, pipelines and filters, client-server and distributed object architectures. The architectural style presented in this paper is a refinement of the pipeline architectural style, as a data feed can be considered as a pipeline. We adapt and refine this architectural style to be suitable to process real-time data feeds on Windows NT platforms.

Architecture description languages (ADLs) have been developed over the last decade in order to describe the functionality of software components. A good recent introduction to the literature on architecture description languages is given by [6]. The most important ADLs proposed in the literature are UNICON [8], Wright [1] and Darwin [5]. The architecture description languages that have been developed so far support the specification of components and the services they provide, their decomposition into subcomponents and the different interconnections between componentents. At the time we started to develop the architecture none of these languages were available and we did not use an ADL for the description of RTD. However, we think it would be a worthwhile exercise to do so.

The development to our system is related to research on real-time systems. Real time operating systems were developed that had flexible scheduling policies so that processes could be given priorities flexibly. Microsoft NT, the system this architectural style is designed for is not such an operating system. This had several consequences on the architectural style that we will outline later.

Microsoft have produced guidelines for creating real time systems for financial market data. These guidelines were published as a Microsoft whitepaper referred to as Windows Open Systems Architecture for the Exchange of Real Time data (WOSA/XRT) [7]. The architectural style introduced in this paper complies with these guidelines.

We have not only defined an architectural style for multiple real-time data feeds, but we have also implemented the style in such a way that it can be customized in many different applications. In order to do so, we have employed the paradigm of component-based development [10]. We have developed coarse-grained components for all generic components of the architectural style and publish a set of interfaces to these components.

Our architectural style for multiple real-time data feeds is implemented using Microsoft's Component Object Model [3]. COM greatly facilitates component based development by providing an interface definition language (IDL) and multiple language bindings. We utilized the IDL to define the interfaces that occur in our architectural style and we used multiple COM language bindings to implement components in different languages.

3 REQUIREMENTS

We determined a number of requirements prior to the development of the RTD architectural style.

1. The architectural style should be capable of integrating feeds of any type of data. The feeds might be proprietary or be provided by commercial data vendors, such as Reuters or Bloomsberg. Examples could be as broad as financial trading data or positions of vehicles that are automatically tracked. The style should provide a mechanism to input these data.

2. The style should be capable of taking data feeds from multiple heterogeneous sources. The style should support the unification and integration of these data feeds.

3. The style should support the filtering of those information from the data feeds so as to reduce the information overload. The style therefore should have a mechanism to customize the filter mechanism to the type of data that is being input to a particular instantiation.

4. The filtered data should be provided to more than one output program, which would display, print or otherwise process the data.

5. The style should support the archival of all data that has been input through the different data feeds.

6. A strategic positioning of the company that developed the style demanded deployment on the Windows-NT platform only. This may seem too radical from an academic point of view, but it enabled the construction of Windows-NT components that implement generic components of the style, while it could still be flexibly customized by adding specific components.

These requirements are not only important for developing the architectural style, but they are also relevant for its use;

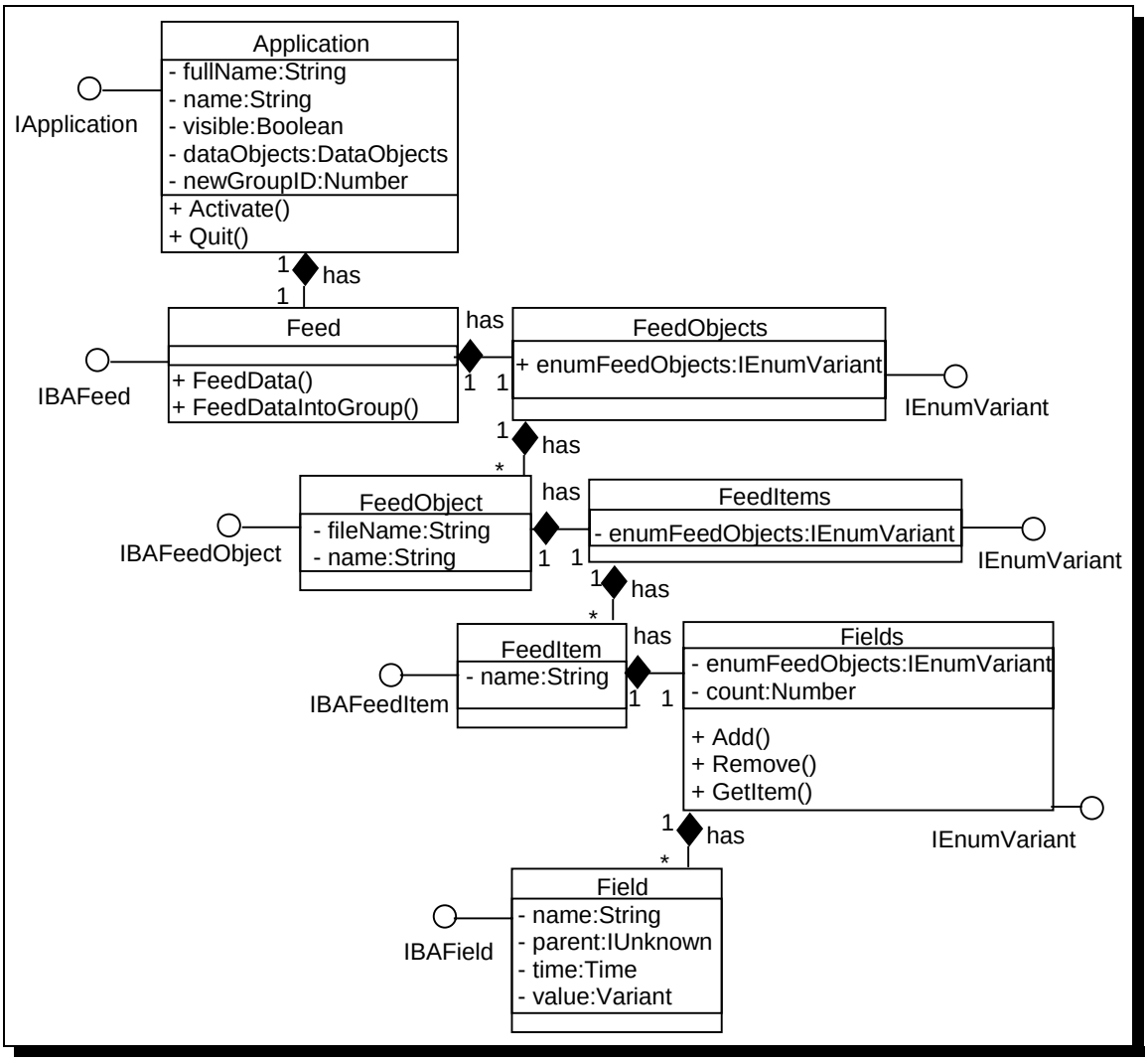

Figure 3: Object Model for Input Component

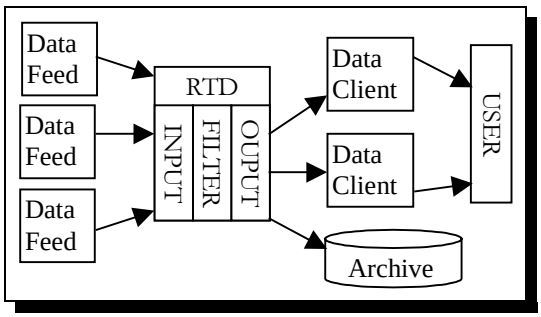

Figure 2: Dataflow through RTD

if a new development effort has to meet similar requirements RTD is a candidate style for the architecture of that system.

4 ARCHITECTURAL STYLE FOR RT DATA FEEDS

The information domain of the system was not easy to define, as there was not a specific goal for this system but rather to provide a generic solution to a problem that had been encountered numerous times previously. All that could be said was that any data entering the system could be placed in this system and then filtered and provided to client programs. Figure 2 presents an overview of the flow of data through the RTD architectural style.

This system would filter the data and only keep those data that the user would require. It would archive the data, it would be prudent to allow the user to choose what should be archived and how often. Depending on the inputs and filtering, huge amounts of data could amass very quickly.

In order to further refine the design each component shown in Figure 2 was refined. Given that the architectural style should be targeted to Microsoft operating systems, we chose to refine this high-level architecture using Microsoft's component object model (COM). Each of the three main building blocks of the architecture is now examined.

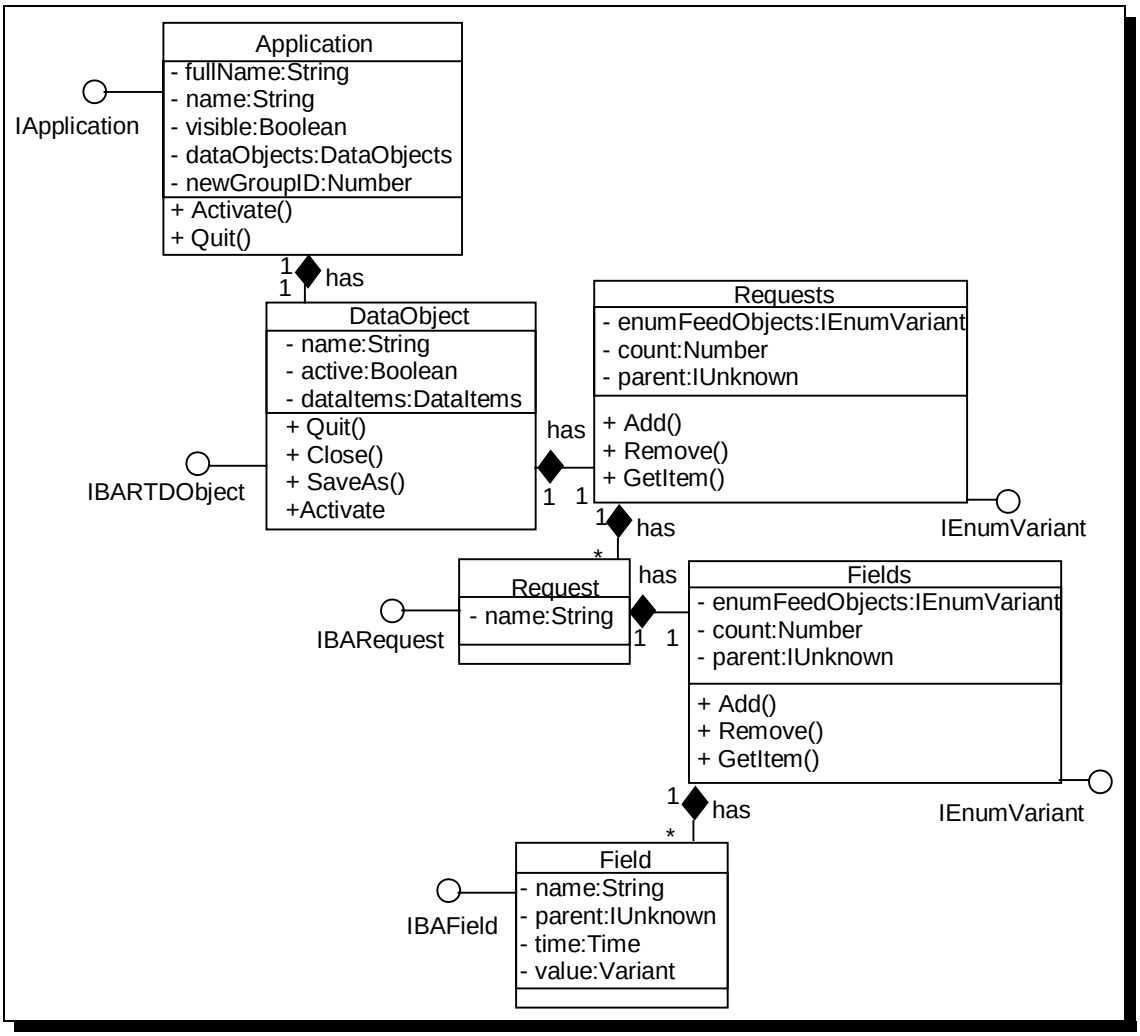

Figure 4: Object Model for Filter Component

For the definition of these components we exploit a number of characteristics that are common to any data feed. A data feed is a sequence of objects that are being fed into RTD. Each such feed object has a set of data fields that are used by RTD to make archival and filtering decisions.

Input
The purpose of the Input component is to collect data from different input feeds, to unify the data and to pass it to the filter component. Figure 3 uses a Unified Modeling Language (UML) class diagram [2] to show the object model for the input component. COM strictly separates interfaces from implementations. We therefore use UML interfaces (that are referred to as lollipops in the literature) to show which COM interfaces an object implements.

The `Application` class is the only RTD input class that can be directly accessed from other, possibly distributed COM component. These are likely to be implementations of data feed components that act as adapters and convert proprietary data formats into a form supported by RTD. The `Application` class within RTD is a singleton [4], which means that there will only one instance of that class. In order to feed data into the RTD system, the data feed components will need to gain access to a `Feed` object, from the Application object. Data can be entered into the system by invoking `FeedData()` and `FeedDataIntoGroups()` from a `Feed` object. The concept of a data group was created in order to allow the feed component to create links between certain types of data. A `Feed` object can get a group identifier from the Application using `NewGroupId()`. A group identifier is a unique number for the RTD system.

Filter
The purpose of the filter component is to filter the data that is passed via one of the input components and to select that subset that users are interested in. The object model for the filtering component is shown in Figure 4. Again, `Application` is a singleton that acts as a root from where the hierarchy of filtering related objects can be retrieved. To request that

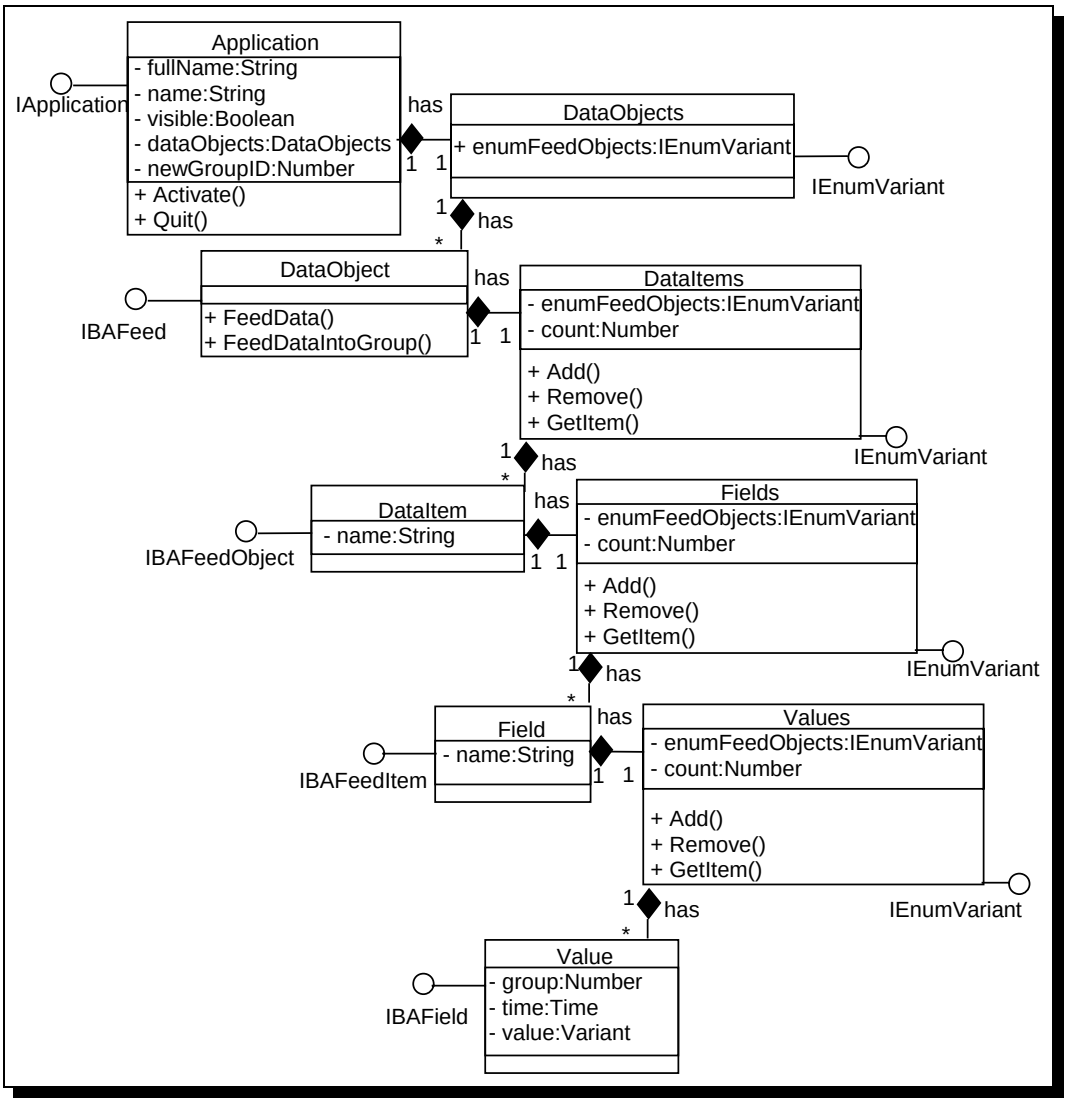

Figure 5: Object Model for Output Component

particular types of data be collected by an RTD implementation, a `Request` object needs to be created within the `DataObject`. The `DataObject` represents a collection of `Request` objects. As an example, a `DataObject` could represent the London Stock Exchange and a `Request` could represent British Telecom shares, a particular financial instrument. Each `Request` object then contains a list of the data fields to be collected. So an example would be to collect the Bid and Offer fields only.

There is a similarity that can be noted between the architecture for the filtering and the input. In particular, the `Application`, `Fields` and `Field` class and their respective interfaces are reused.

Output

The purpose of the output component is two-fold. Firstly, the output component provides the filtered data to Data clients. Secondly, it archives the filtered data for later use. The diagram in Figure 5 shows the object model used by the output component. The structure again reflects the aggregation hierarchy that we exploited also in previous two components. The `Application` object acts as the single root from which the hierarchy of available objects can be accessed. The `DataObject` class will be the same as the one used for Filtering. If we request for data in the London Stock Exchange `DataObject`, it will be the same `DataObject` as the one used to access the data collected for the London Stock Exchange. The `DataItem` object represents the actual item that RTD is collecting the data for, so an example again would be British Telecom. This `DataItem` then contains a collection of `Fields`, these `Field` objects are from the same class as the `Fields` presented in the Input and Filtering components. The difference is that now the field can contain a list of values to represent data ticks from the feed.

Exploitation of COM in RTD

In this subsection, we note how the RTD architectural style exploits Microsoft's Component Object Model. RTD explicitly uses the separation of interfaces and implementations that is demanded by COM. It also exploits the ability to deploy components within processes as dynamic link libraries (DLLs) or as separate executable processes (EXE). Finally, the fact that RTD output has a COM interface enables seamless integration with other standard COM packages, such as applications of Microsoft's office suite.

RTD strictly follows COM's principle of separating interfaces from implementations. Figures 3-5 have shown the interfaces that are defined as part of RTD. These interfaces have generic implementations that were also shown. For customizations of the architectural style, however, it is possible to replace a generic implementation of an interface with a customized one.

It is transparent for the design and implementation of components whether they are deployed as DLLs or as EXEs that may even run on different machiens. Microsoft's interface definition language compiler generates the client and server proxies needed for (distributed) inter-process communication. COM deploys these proxies transparently if a transition is made from a DLL deployment to an EXE deployment. DLL deployments generally respond very fast because operation invocations are implemented as local method calls. EXE deployments on the same machine have the benefit that processes can be launched and terminated independently. With EXE deployments of data feeds or data clients, failures in feeds or clients do not affect the server. The same holds with deployment as EXE servers on different machines and distributed deployment adds load balancing capabilities. The fact that the type of deployment is transparent to the design and implementation of the components allows deployment decisions to be made by administrators and makes administration of RTD based systems very flexible.

The fact that all RTD components have COM interfaces also enables a smooth integration with other commercial off-the-shelf software that use COM. In particular, we have integrated RTD output components with Microsoft Office components that are used to manipulate the integrated and filtered data feeds further. To facilitate this integration, we have exploited the ability to extend Office products using Visual Basic scripts and the fact that every COM component and therefore also the RTD components can be accessed from these Visual Basic scripts.

Soft Real-Time Constraints in RTD

5 INSTANTIATION OF RTD STYLE

6 EVALUATION

RTD has been further instantiated in the Black Ace Software Engineering product named BASE Market Monitor. This product tracks share prices and news from financial markets. Feeds have been written to accept data from Datastreams Market Eye, Teletext and World Wide Web pages. Features of RTD have also been used within Cognitechs Market Surveillance System and also more recently in the Visual Global Markets product. These products use more 'heavyweight feeds from ISMA and Reuters respectively. The BASE Market Monitor product is aimed at smaller investors and home users. The feeds that is connects to themselves provide either 'soft real time or time delayed data. This product runs on a single machine and the RTD 'engine is a single instance application. Many clients packages then talk to this 'engine in order to provide different views of the data to the user, such as tickers, charts and price screens.

The architectural style holds up well to this model. If a single client fails, it does not bring down any other clients or the RTD engine, as they each sit in their own protected process spaces.

The response times and therefore performance of the system depends heavily on the activity of the feeds. When the market is busy, the load increases and the response time drops. Actual timing measurements have not been taken but existing users of the system seem content with the performance, and we are the first to know when there is a problem!

The Market Surveillance System required a 'harder real time system. The surveillance team in a stock exchange needs to know within seconds if market makers or traders are not adhering to the rules of the market place. Initially this system was written with the European Exchange EASDAQ where ISMA provide the TRAX data feed. By collecting the data on a server machine the RTD system was extended to provide many client machines with the real time data. The classical client-server model has been used for this system and it is currently being used by the surveillance team within EASDAQ.

Cognitech then decided to take the architecture further in the Visual Global Markets product. This product takes full advantage of the DCOM model and is based on a three tier architecture. The data provided by Reuters enters a server machine, where it is also archived. This database server advises a second server of data changes. This second server has a set of complex filtering and calculating components. These middle layer components apply the business rules to the Reuters data. Several client machines then can be advised of information that is of use to the end user. To date two different versions of the client have been written. One which is a fully functional program for filtering, sorting and editing the data, and the other which is an add-in to Microsofts Excel application, where the data can be further manipulated. It is this ability to link to such an industry standard as Excel which makes the RTD architecture so powerful.

7 SUMMARY AND FUTURE WORK

REFERENCES

[1] R. Allen and D. Garlan. A Formal Basis for Architec-

tural Connection. *TOSEM*, 6(3):213–249, June 1997.

[2] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[3] D. Box. *Essential COM*. Addison Wesley Longman, 1998.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software*. Addison Wesley, 1995.

[5] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schäfer, editor, *Software Engineering — ESEC '95, Barcelona, Spain*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer, 1995.

[6] N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Software Engineering - ESEC/FSE '97, 6^{th} European Software Engineering Conference, Zurich, Switzerland*, number 1301 in LNCS, pages 60–76. Springer, 1997.

[7] Open Market Data Council for Windows. *WOSA Extensions for Real Time Market Data (WOSA/XRT)*, ????

[8] M. Shaw, R. DeLine, D. V.Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *TSE*, 21(4):314–335, April 1995.

[9] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[10] J. Udell. Componentware. *Byte*, May 1994.