A Modular Framework for Reusable Research Software

by

Patrick William Anderson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000
[June 2000]

Author .
Department of Electrical Engineering and Computer Science
May 18, 2000

Certified by... .....
/ Nancy G. Leveson
Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Modular Framework for Reusable Research Software

by

Patrick William Anderson

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Research which requires developing software is burdened with the effort of developing tools to demonstrate and evaluate various topics and approaches. Often such research tools are very short-lived, giving way frequently to new tools which demonstrate new ideas and directions of the research. The result of such continuous change to the requirements and features of reasearch tools is that they are usually discarded and new tools developed when new needs arise. This thesis presents a framework on which research tools can be constructed which maximizes the reusability and maintainability of that code, and can reduce the cost of new development.

Thesis Supervisor: Nancy G. Leveson
Title: Professor of Aeronautics and Astronautics

Acknowledgments

I would like to thank the following people for their assisstance, support and encouragement on this project:

My supervisor, Professor Nancy Leveson, for taking on a student she need not have and providing me with her support and encouragement in the face of adverse and difficult circumstances. Thank you very much. I am most grateful.

Jeffrey Howard for providing me a sounding board, mercilessly attacking my ideas at every turn, and generally giving me valuable feedback on the project and this thesis while they were in development. His assisstance was essential to this effort.

Michael L. Smith, for first opening my eyes to the world of software engineering and design so long ago.

Contents

Chapter 1

Introduction

Research software is a particularly challenging software engineering problem. It is functionality driven, as is all applications development, but the requirements are continually changing as the research progresses. This dynamic and very limited environment inevitably leads to a host of unmaintainable and barely functional, software which must be discarded with each new advance.

Ideally then, research software would be supported by a body of code which incorporated basic functionality most often used by the code so that changes could focus on research driven needs and not on the detailed aspects of implementing those needs. In principle, this is something that every body of research software would have, but in practice limits on time, money and personnel drive development to be as rewarding as is possible at all times, which implies that time spent doing maintenance to avoid a greater amount of maintenance in the future gets sacrificed in the name of greater functionality.

While there is very little work dealing with the causes behind the situation as it stands in the academic community, anyone familiar with conditions there vis-a-vis software development will find my description uncomfortably familiar. This author regrets that only a vanishingly small amount of research into the real processes which are used in practice in academia to develop software exists. Perhaps this situation will be rectified in the future, but it seems unlikely.

The purpose of this thesis is to present a design for such a body of code and argue

that the design meets its goals. These goals include minimizing as much as is possible the cost of modifying the software which is supported and not significantly adding to the overall development time of the research software in question.

1.1 Rapid Changes in Requirements

The requirements of research software tend to change often and at times drastically. This rate of change is driven by the progress of the research and changes in its focus. This is only natural, as the purpose of the software is to assist (or even enable) the research, and thus as the research progresses, the needs it imposes on the software must change. This sort of phenomenon is especially evident in computer science related research, where a research compiler might be built to support a research language, but need to support a new version of that language on a regular basis, as often as on a weekly basis. This is not to say that this sort of phenomenon is limited to computer science research, as any research that requires computer assistance often suffers from analogous software requirements problems.

1.2 High Turnover Rate

Research software tends to be replaced often, resulting in wasted effort as new implementations of the same functionality are built from scratch over and over again. This can be caused by "Not Built By Me" syndrome among the programmers as staff changes as easily as by other more rational problems. Quite often, the software simply reaches, or is delivered in, an unusable state, or an impossible to maintain one. The end result is that quite often discarding the software and starting anew is obviously easier than attempting maintenance of it.

1.3 High Development Costs

The usefulness of research software to the research effort is directly related to the usability of the software in question. The functionality of the software is of course paramount, but if that functionality is not usable by the researcher then it might as well not exist. This leads to a situation where the only useful effort on the project is to add new functionality.

This push for usable functionality often leads into the realm of graphical user interfaces (GUIs), which are especially costly to develop as they involve a great deal of minutiae to be dealt with and a large code base to be really usable. An under-developed and buggy GUI will alienate the users more than a less functional but usable alternative. In fact, the current set of tools that this thesis work is designed to replace are so troublesome that every requirement from the users for the new tools dealt with usability issues instead of functionality.

1.4 Obstacles to Reuse

Since the overriding pressure on the development of research software is the production of useful and usable artifacts, it is usually under-designed and sloppy. Whatever option works for the current need and can be implemented most quickly will be the one chosen when design decisions come up during development, with no thought to future needs. Since development staff and time are at a premium, refactoring will always be pushed back in favor of adding new functionality, inevitably resulting in a mangled design and unmaintainable code. No one can argue that an undirected design effort will result in unmaintainable code, but usually this problem is correctable as it happens through what has come to be known as refactoring[5]. Unfortunately since the time to refactor is not justifiable, then refactoring will not be done.

All of this adds up to code which is highly interdependent, ill documented, and generally a mess. Attempting to reuse even simple and well documented code is sometimes more work than it saves[3], so it is no surprise that reuse is not common

with research software. A code base is abused until it can no longer reliably serve its intended function or a staffing change occurs, and then scrapped in favor of rewriting the whole system from scratch.

Chapter 2

System Requirements, Safety and the SpecTRM Toolkit

The focus of this thesis is a specific application: the Specification Tools and Requirements Methodology (SpecTRM) toolkit. The toolkit enables researchers to explore the benefits of using a standard language for specifying systems. The goals of the SpecTRM Requirements Language (SpecTRM-RL) are to provide a readable and analyzable specification language for use in requirements and specification documents.[8] The purpose of this project was to design a robust modular framework for the toolkit with the goal of reducing the effects of changes in the specification language and its visual representation on the overall code base of the toolkit.

The problem of producing reliable safety-critical systems is not a new one. Indicators exist that suggest many safety-related accidents are caused by system faults traceable to deficiencies or errors in the system requirements or specifications.[10] Intent Specifications and the SpecTRM toolkit seek to address this problem.

2.1 Safety and System Requirements

More and more, computers are entrusted with the operation of safety critical systems. Unfortunately, software engineering is at best an inexact art, and so these systems are extremely difficult to build. Testing cannot feasibly be done such that all possibilities

are examined, nor can such critical systems be allowed to fail. As it turns out, many critical systems fail not because of software errors, but due to incorrect software. The distinction is subtle but important: the software does what it is supposed to, but what it is supposed to do was incorrectly communicated to the software engineers.

Specifications that can be read and understood by all of the individuals involved with developing a system is key to solving this problem. Professor Nancy Leveson suggests a new form for specifying systems, the intent specification. The intent specification[11] differs from a usual specification in that it is a hierarchical abstraction of intent instead of just an abstraction of a requirement and how to fulfill that requirement. The hierarchy comes from the direct linking of each level to the level below it to indicate the origin of each element in the lower level. This introduces trace ability throughout all levels of the specification so that ideas at one level are clearly connected to their progenitors at the level above. Level one abstractions define the goals of the system, level two the design principles, level three the black box behavior, level four the design representation and level five the implementation representation.

2.2 The SpecTRM Toolkit

The SpecTRM toolkit serves to explore ways in which a specification language can improve the usefulness of specification and requirements documents. While the current focus is on verifying the completeness of specifications, the properties of SpecTRM-RL make it very useful to the requirements and specification development process in a number of ways. The toolkit must support the language and allow for easy experimentation into other advantages of the language, as well as adapt easily to changes in the language as it evolves.

As it is currently conceived, the toolkit provides means to edit, analyze and simulate SpecTRM-RL, as well as prepare level one and two intent specifications. SpecTRM-RL is designed to convey level three intent specifications of black box system behavior.

2.3 Toolkit Design Criteria and Requirements

Since the driving motivation behind a new development effort for the toolkit was to produce an application that did not suffer from the problems of the existing tools, the design criteria and requirements were simple. The new tools should be easy to use and maintainable above all else, as these are the two major flaws in the existing tools. As far as requirements are concerned, a duplication of the existing functionality was the most immediate concern. Also, it was determined that due to the rapidly evolving nature of the language, the development effort would have to focus on keeping the code base very flexible and maintainable so as to be able to keep up.

All user requirements have been related to defects in the existing toolkit, so the design of this set of tools was mostly informed by examining how the users used the current toolkit, that toolkit's features, and research papers on new versions of the language and possible avenues of future research.

2.4 Design Goals

The guiding principles for the design of the toolkit were:

- Flexibility
- Extensibility
- Maintainability

Many of the design decisions have been motivated by examination of existing applications, application frameworks, and foundation class libraries. While even an improved reproduction of any existing code base is the object of this project, existing projects have provided valuable insight on the features best shared by parts of an application, as well as features that best enable the isolation and coordination between parts of an application necessary to give the properties desired in this tool set.

The practical result of these goals is a modular framework as described in Chapter 3. The principles guiding the design of such frameworks are described there, and the design itself is detailed in Chapter 4.

2.4.1 Flexibility

The goal of this project is to produce a framework that supports a great deal of flexibility and extensibility for whatever application is using it. In that spirit, the framework only specifies operations necessary to its operation, and avoids specifying behavior or functionality that might normally be associated with the components described so as not to place unnecessary limits on their design.

However, the purpose of this project is not to produce a fully general framework, but one that facilitates the SpecTRM tool set. Thus the framework attempts to harness as much generality as is possible without being overly general, or attempting to generalize features that would not provide benefit to the SpecTRM tool set. Generality is present in the interest of increasing flexibility. Excessive generality is avoided by attempting to make the use of the framework as easy and straight forward as is possible.

2.4.2 Extensibility

Extensibility allows new functionality to be added to the tool set without necessitating changes in the existing code base. The primary means by which this is accomplished is to make as much of the framework as is possible loosely coupled, if not completely orthogonal. In terms of both functionality and implementation, decoupling of parts of the framework prevent changes in one part from affecting others.

The practical upshot of loose coupling is fixed messages and call chains without fixed recipients. This enables delegation and replacement of default handlers within the framework so that any given module can be replaced with a module that implements that interface differently, or new modules can easily be added to implement wholly new functionality.

2.4.3 Maintainability

Maintainable program code is essential to this project if any of the other goals are to be met. While the basics of maintainable code, readability and good documentation are tied almost exclusively to the process used to generate the code, higher order aspects of maintainability can be designed into the system. Once the basic needs of maintainability are met, maintainability becomes a function of the design, architecture and testability aspects of the system. A well designed system is flexible and easily understood if well documented, so changes to the system have minimal impact and are easy to integrate. A good architecture for the system is also essential simply because a well designed but poorly implemented system is no more maintainable than a poorly designed system. It might even be argued that the architecture is more important than the design, because the implementation can work around many design issues, while the design cannot possibly make up for a poor implementation. Finally, a system is only maintainable when it can be regularly and easily tested at every level so that the effects of changes to the system are immediately obvious. The maintainability of the toolkit will directly determine its lifespan as a useful body of code.

2.5 Existing Tools

The current implementation of the SpecTRM toolkit is based on a large set of macros for the Microsoft Word word processing application, and a number of external programs to implement the simulator and visualization aspects of the toolkit. The primary interface to the system is through the Word application, where users write the majority of the specification using special templates and custom macros. The simulator is a separate program, called Nimbus, that uses an older version of the language, the Requirements State Machine Language (RSML), to simulate the specification after it has been translated from the Word document. The simulator is highly dependent on the Component Object Model (COM) as all external event sources and sinks communicate with the simulator via COM channels. The system as a whole is

very fragile, and requires a good deal of effort and knowledge to use.

2.5.1 Nimbus

The Nimbus system is a set of tools for creating intent specifications and simulating SpecTRM-RL specifications. The various major tools are covered below:

The SpecTRM-RL Editing Macros

The SpecTRM-RL editing macros automate the process of building up SpecTRM-RL specifications, and are represented to the user in Microsoft Word as a series of text labeled tool-bars. Each button represents a language construct, essentially a keyword, and causes that keyword to be inserted into the document at the current caret position.

The macros do no consistency checking or other assistance with the programming process. The user is required to write what superficially resembles a coherent document but is in fact a program in SpecTRM-RL.

When the specification is complete, the user compiles it into RSML for simulation, using a macro that creates a new document and populates it with RSML code. This process is very time consuming, even for small specifications.

The RSML Simulator

The simulator program used in the system is a slightly modified version of the original RSML simulator. The original simulator was used with minor modifications to minimize development time of the system. This decision caused many of the problems with the current set of tools.

The simulator presents itself as an application that can load RSML documents, and set up events, inputs, outputs and the like for the system. The simulator also displays a graphical representation of the system's state and modes that is synchronized with the actual state of the simulation as it progresses.

The Channel Manager

The channel manager allows the user to set up input and output channels for the simulation. The simulator provides both input and output channels according to the system being simulated, but channels can also be provided by any application supporting the COM protocols. Channels are accessed individually, and any number of inputs and outputs can be arranged, as well as observers. For any given simulation run for each channel an input, an output and optionally an observer are chosen. This design allows multiple possibilities for sources, for instance, when simulating the system under consideration, without having to set everything up multiple times.

2.6 Problems with the Current Solution

The current implementation of the SpecTRM toolkit suffers from several usability and functionality problems necessitating a new toolkit. The most glaring is the lack of documentation and coding standards, which makes the code unmaintainable. Other issues with the toolkit are summarized below.

- The compilation from specification to the form used by the simulator is a very time-consuming process, on the order of hours for even moderately sized specifications. Performance this poor makes the system essentially unusable.

- The compilation phase returns no syntactic or semantic errors until the compilation is complete, as checking of the compiled specification is done by the simulator when it is loaded. Even simple syntax errors result in a complete re-run of the compilation phase, which as noted above can easily take hours.

- Users of the system must write code in the raw specification language in order to make the specification compile. The reasoning behind the decision to require the user to write code is unclear. Additionally, the language syntax is confusing and unnecessarily verbose. Users should not be exposed to the internals of the toolkit, and doing so requires a level of knowledge that is unrealistic to expect among practicing engineers.

These first three difficulties have led members of the research group to translate multi-hundred page specification documents by hand into the simulation language rather than attempt to correct the specification to the point where it will compile. This sort of activity is entirely unnecessary, as the purpose of the toolkit is to automate using the specification language, not make it more work intensive. Also, from a practical standpoint, human translation is error prone and simply not a good long-term solution.

More important than these are the more technical issue that contribute to the ineffectuality of the current software as a tool and its unsuitability as a long term solution.

- Basing the toolkit around the Microsoft Word word processor incurs a huge overhead, as well as subjecting the toolkit to the instabilities present in that product. Empirical evidence from the group suggests that documents over a certain size cause fatal errors in the application in most cases, and are often corrupted in the process. Since the templates and macros that make up that part of the toolkit are already sizeable, this makes creating even simple specifications a risky proposition.

- Dependence on other features of the Microsoft platform, such as the Component Object Model (COM) and other Microsoft applications makes the toolkit very high maintenance, as many different versions of these technologies are in common use.

- Fundamental limitations of the current implementation prohibit the system from being useful outside of the research group, the most important of which is performance, as detailed above.

It is not any one of these deficiencies that necessitated replacing the existing software, but the overwhelming negative impact of all of them, exacerbated by the condition of the code and documentation for the system. The next chapter deals with the design of the foundation for the new toolkit in a general way to help illustrate the actual design as presented in Chapter 4.

Chapter 3

The Design of Modular Frameworks

The design of a modular framework is a careful balance of generality and domain-specific needs. The right amount of generality for the application must be chosen carefully, and the design must reflect this decision. Too much generality, and the benefits of the framework are overwhelmed by the extra work needed to use it. Too little, and the extra effort spent on design and construction of the framework is wasted as the framework has a very limited lifespan.

The purpose of a modular framework is to minimize work necessitated by changing requirements. The more stable the foundation of an application, the more the application code can rely on it. The more modular the application is, and the more orthogonal those modules are with regards to each other, the more flexible the application is, and more importantly, the more it withstands changing requirements. So the design goals of a modular framework are to provide a stable foundation and enable easy modularity for the benefit of the application.

It is important that using the framework be easier than simply using the underlying services it abstracts directly for two reasons: it will be less likely that wayward programmers will bypass the framework, and it makes little sense to create a construct whose purpose is to reduce workload that requires more work to use than the work it saves.

3.1 What is a modular framework?

A modular framework is an architectural pattern that provides services with which software can share common functionality and consistent features without limiting the scope of future development. A modular framework usually consists of a mechanism that enables modular software components and a set of additional services for use by the application. The intent is that the component system and services have very static interfaces and that functionality is added by adding components or services to the framework instead of attempting to extend the interfaces of existing components or services.

Modular frameworks have one goal: to isolate the application from changes in underlying services and individual components of the application from each other. This goal is achieved in two ways. Modularity encourages discrete functionality to be encapsulated in separate components or modules, and it promotes orthogonality and reuse. A framework provides policy-free services, not application functionality. It is, instead, an environment in which application functionality is constructed.

Modular frameworks bridge the gap between support services and application code in order to stabilize and abstract the environment in which applications operate. Even the touted portability of Java isn't enough, because the problem stems not from the presence or absence of abstracted services, but instead which services are abstracted and to what degree. Applications have specific needs from their services, and the abstractions of those services that they use must be tailored to their needs. Graphical User Interface (GUI) libraries, often mislabeled as "foundation classes[1]" and component systems do little to actually provide a basis upon which applications can build. They provide essential services, but to mistake those services for a framework upon which a stable application can be implemented and grow is a dangerous design error.

Most applications share a large number of common operations, especially in areas

[1] As we shall see, a very different definition of this term from the norm is used here, one which seems to better capture its intent.

such as data and settings management. Most GUI libraries, most notably Microsoft's Microsoft Foundation Classes (MFC) and Sun's Swing Toolkit, provide such services. This seems to also be a design error (on the part of the designers of those toolkits), as it tends to couple these services to a single fixed application architecture, which is usually heavily biased towards the toolkit in which these frameworks appear, both in terms of the actual widgets of which the GUI toolkit is formed, and the application and document model favored by the toolkit. It is nearly impossible to use user constructed widgets with the provided framework, and even more difficult to use a different application model while still taking advantage of the provided services. For a concrete example of this, in Sun's Swing toolkit the TreeModel class (the model portion of the abbreviated form of Model-View-Controller which Swing is based on), assumes that the children of a node will be stored in an array. Java supports a wide variety of collection classes, but the children of a node must be stored as an array, and there is no practical way to replace the model used by a view. More importantly, the mechanisms at work in an application should be entirely separate from its visual presentation, so overloading a visual toolkit with that sort of functionality seems a case of feature creep. The modular framework allows all facets of a set of tools or application suite to share common settings and data management functionality, as well as common functionality that is domain-specific. For example, a program development suite might share a module that provides parsing services for use by the compiler, editor and code analysis tools.

A modular framework is distinct from component systems such as the Common Object Request Broker Architecture (CORBA), Component Object Model (COM), and Java Beans in that it is always domain specific to some degree. These other solutions may provide several of the mechanisms of a good framework, even mechanisms which are necessary for the implementation of a good framework, but they are not the whole story. Just as bare file access does not constitute a useful data management mechanism to the application programmer, a component system in isolation does not provide the domain-specific but application-wide functionality needed to really reduce programmer overhead during application development. These component sys-

tems provide easy and standard communications mechanisms between components, but none of the domain specific services necessary to meet the goals of a good framework.

Consider a component system, such as the Common Object Request Broker Architecture (CORBA). It provides a very useful communications and marshaling system for creating both local and distributed software components. Now consider an application using such a mechanism. The application must add its own error checking, consistency checking, and usually build a basic set of components to be able to realize any functionality. The goal of a modular framework is to adapt this mechanism (or something like this) into something that an application in a particular problem domain can use almost immediately.

Neither is a modular framework simply a generalization of common functionality that may be shared by several modules and usually is the impetus for class hierarchies in object-oriented design. Generalization suffers from the same fatal flaw as statistics: The trend is only as good as the sample space from which it is derived. The tendency to generalize object hierarchies in general is a good one. Unfortunately, usually this overgeneralization results in hierarchies which are very brittle — as new examples of the classes generalized over are produced, the hierarchy as a whole must be adapted to accommodate them.

Consider a visual component for displaying graphs. It might contain a drawing hierarchy, where all visual components of the graph inherit from an abstract class that defines the interface for composing graphs and drawing them. In this domain-specific graphing hierarchy, adding a new type of edge would necessitate refactoring the inheritance hierarchy at least to the point of adding a new base class for edges. This involves modifying the existing Link class, and adding two new classes, one for the abstract parent, and one for the new edge type. Even new classes that already have abstract parents can disrupt the inheritance hierarchy a great deal. Suppose that the draw method assumes a graph representation which is tree based. Suppose further that a new class is introduced that allows the construction of arbitrary graphs. Potentially every class in the hierarchy must be reworked, if the abstract classes contain

any of the logic for drawing the graph which makes assumptions about its structure. While this example may seem far fetched it is based on one simple assumption: We cannot anticipate future needs when making functional generalizations.

Modular frameworks are also sometimes referred to here as foundation classes. It seems that this is a better fit for the term than the GUI frameworks to which it is usually applied although those frameworks sometimes include a small portion of the basic functionality of a modular framework. Examining the phrase, it seems that a class hierarchy that provides a stable basis for implementing the functionality and visual presentation of a set of applications or tools better deserves the name "foundation classes" than does a GUI toolkit with some data management features mixed in. A modular framework provides the stable foundation needed to develop a flexible and robustly designed application suite or tool set. While it is folly to expect the world at large to conform to this author's notions of propriety in such things, this is the definition of that term that will be used in this paper.

3.2 Issues That Impact Framework Design

For the purposes of this discussion, changes in the needs of applications are divided into four broad categories: system requirements changes, application requirements changes, visual requirements changes and abstract requirements changes. System requirements include all requirements that involve underlying services, the operating system, or system libraries. Application requirements address what functionality is present in the application. Visual requirements are all requirements of the system that deal with the presentation of the system to the user. Abstract requirements are those requirements of the system that do not correspond directly to any one aspect of the system and that are not generally as concrete as the other forms of requirements. These four categories can be considered to divide all the requirements of the applications based on the framework into four sections that layer hierarchically.

The purpose of the framework is to mitigate the effects of changing these requirements as much as is possible and to minimize the amount of code thrown away no

matter how drastic the changes may become.

3.2.1 Addressing Changing System Requirements

The purpose of the framework being a strong foundation for the application is to isolate the application from system changes, and even system requirements changes. This goal is accomplished, in broad terms, by isolating and abstracting system functionality for the application. The real issue is how to go about accomplishing this amazing feat.

In a naive view of the problem, it might be sufficient to simply abstract system services. This abstraction promotes portability, but not much else. The key is not to abstract the services that are provided, but to abstract the services that the application requires and work from there. The important distinction between this approach and simply doing a top-down design of the application is that instead of starting with the features of the application and designing from there, one starts with the needs of the features of the application and then applies a top-down approach to the design.

For instance, almost every application does some form of configuration and settings management. In many cases this process is as simple as maintaining a single file with a list of key/value pairs. But it can be as complicated as the Microsoft Windows Registry. In the abstract, these two extremes are not at all different — in reality, the reality of implementation, that is, they could not be farther apart. By creating an abstract settings management service in the framework, we relieve the application programmer of a good deal of repetitive work.

How should this be accomplished? Naively, we might simply create an interface that defines a key/value pair mapping associated with a file. This technically fills the need and abstracts the underlying system, but in actuality turns into a greater amount of work for any settings management more complex than a single flat hierarchy sharing a single location on disk. What if a hierarchical system is needed? Or a conditional system based on scoping of some sort? The naive model quickly breaks down and becomes much more of a burden than a boon.

A better solution would be to approach the problem from a top-down standpoint

— what does the application need to concern itself with? The application (and by extension the application programmer) only wants to be able to store structured, persistent state. A good abstraction might be a language-level environment model for settings. As long as this abstraction is fairly opaque and has good defaults for storage location and structure and the like, then the application need never concern itself with changes to how it operates.

It can be seen that by choosing abstractions in the framework carefully and paying particularly close attention to their scope, that a robust and reliable foundation for application programming can be created. The services provided may be domain neutral or specific, but they should all be as orthogonal and limited in scope as is possible. To paraphrase Antoine de St. Exupery, a good framework service abstraction is arrived at, not when there is nothing left to add, but when there is nothing left to take out.

3.2.2 Addressing Changing Application Requirements

Application requirements usually develop and change significantly over time. I do not refer here to so-called "features" of an application, but to the original requirements that bring about the concrete representations in the software that are labelled as application features. In fact, the literature dealing with office automation suggests that this change is entirely necessary to that process, and that good business automation begins by translating the forms for a business process onto the user's screen and then getting feedback from the user about how the process might be improved from there. This is due to the fact that the possibilities inherent in automation are not obvious to the user until the process has been automated, and at that point the system usually turns out to need a different set of features[1]. In fact, it is often lamented in the literature that most systems are delivered implementing a set of features that are entirely unlike what the users desire, largely due to this phenomenon.

Modular frameworks address this issue by making it easy to modularize the functionality which provides the features of an application. With this sort of modularization in place, it is much easier for an application to adapt to changing requirements,

as changes simply introduce new modules, obsolete modules, or require changes to how modules are used.

So, by reducing changing features to changes in discrete modules, and by making the construction of modules simple enough to encourage systems with a great number of them, distinct and abstract, a modular framework reduces the design and work impact of a rapidly evolving set of requirements.

3.2.3 Addressing Changing Visual Requirements

While modular frameworks do not explicitly address the issue of changing visual requirements, they do allow visual components to be isolated from functional components and vice versa. They also can allow visual components to easily adopt a differing data model than the rest of the system to further isolate visual requirements from other requirements of the system.

The advantages of visual components of the application using a different data model than the functional components are myriad. Applications are discouraged somewhat from exposing the programmer's representation directly to the user, as they are already using a different data model. The visual component is free to change its data model without incurring any more than the cost of updating whatever code does translation between its representation and the system representation. Having two or more distinct representations can aid in catching malformed data structures.

While it might be argued that the cost of maintaining two data structures outweighs the benefits it would bring, this is not necessarily the case. It all depends on the granularity of the modules in the system and their interdependencies. If the visual component is mostly self-contained and monolithic, then maintaining its own data representation is less expensive as conversions need to be done less often. Otherwise it may be prohibitive to maintain such a duplication, but then all components may be exposed to changes in data representations needed by changing visual requirements. This does not necessarily invalidate the claim that the framework will help mitigate the effects of such requirements changing, as it will not always be necessary to change representations to meet the new requirements, especially if the representation is well

designed.

Even without having a distinct and separate data model, visual components that rely on other modules instead of being monolithic incur much less overhead in the face of change. In a monolithic system, the effects of changes are much harder to predict and control than in a more modular one.

3.2.4 Addressing Changing Abstract Requirements

Abstract requirements are a difficult concept to deal with concretely. They have been called many things by many different design methodologies, but in essence they boil down to one class of needs: they are those needs that affect the system as a whole and are not reducible directly to some set of other types of requirements. Examples include efficiency requirements ("It should be fast enough to meet our real-time needs.") or interaction requirements ("It should be easy to use."). These sorts of requirements affect the system as a whole and can skew the system's design.

Modular frameworks do not attempt to address these sorts of requirements directly. However, the modularity the framework encourages in the application tends to isolate the functionality most affected by abstract requirements changes. For example, computationally expensive operations can be abstracted into a separate module so that optimization of those operations can be done without disrupting the rest of the framework.

3.3 The Advantages of Modular Frameworks

Modular frameworks are not a silver bullet[6] by any means, and in many cases trying to turn one into the cure for all a project's ills will violate its design goals and turn a potentially helpful architectural tool and the application it supports into a maintenance nightmare. The key fact here is that modular frameworks do not enforce better application design in all cases, but ideally they do encourage it by making the right thing to do the easy one.

Modular frameworks offer many advantages that may be obtained with other

methods, sometimes even with less effort. However, no other method offers the combination of advantages offered by modular frameworks without significant disadvantages as well. Inheritance hierarchies, for instance, offer many of the same advantages as a framework, and while it might seem that a well designed inheritance hierarchy is a sufficient substitute for a modular framework, this is not the case in a research environment. In the research environment, staff turnover is high, and so having a black-box abstraction eases the learning curve for new developers. Inheritance hierarchies are too difficult and time consuming to understand, and too easy to abuse. Even their proponents admit that inheritance-based object designs are easier to abuse and harder to learn[4], and justify their choice by claiming that a white-box abstraction is sufficiently advantageous to a black-box abstraction to make these disadvantages worthwhile. Unfortunately, this is simply not the case. Program code, while it often offers insight into how a system works, only encourages dependence on implementation details.

One need only look at the problems encountered by major operating systems vendors in trying to maintain backwards compatability to find the flaw in this logic. Any information revealed to the user of an API will have to be maintained as true indefinitely at the risk of destroying backwards compatability. Since the goal of a modular framework is to reduce overhead in maintenance and extension of the application, then a black-box abstraction as is present with a component system is preferable to a white-box abstraction where programmers can easily become dependent on implementation details.

Modular frameworks offer advantages in design, development and testing over other methods. While these advantages are not individually unique to modular frameworks, modular frameworks provide them all in a way that is more useful than other methods.

3.3.1 Design

A modular framework allows the developer to design for the general case and still implement the simple case, so functionality develops more quickly and there is still

room to grow. The framework should encourage good modularity and very stable interfaces. The encouragement can affect even the application built on the framework, by serving as a positive example.

A good example of the success, from a design standpoint, of a modular framework is the Quicktime software development kit from Apple Computer. Quicktime has been in production for over 10 years at this point, and applications written for version 2.0 still run on version 4.1.[2] Quicktime supports a robust component model, and instead of changing existing interfaces has always added new interfaces to take advantage of new capabilities. The component interfaces are extremely stable. The only way one might improve on this model would be to use explicit versioning in the interfaces to prevent developers from accidentally using outdated interfaces. Quicktime avoids the usual negative effects of multiple interfaces[3] by never overloading an interface through multiple versions but instead adding new calls when newer functionality is needed. This is not to say that Quicktime is a fairy tale of perfect growth over time — it too has suffered from needing to maintain compatibility with existing flaws, for instance. Quicktime does seem to have weathered the years much better than most of its surviving contemporaries.

3.3.2 Development

The time to a working prototype is greatly shortened when using a modular framework as core functionality can be fully developed without waiting for less central functionality and without hindering the development of such functionality. By carefully implementing the minimal core functionality of the framework services and base components while designing robust and flexible interfaces, effort can be focused on reaching important early functionality goals. As the full flexibility or feature set of those services are needed, they can be fleshed out without disturbing the rest of the development process.

[2]The transition from version 1 to version 2 altered the component interface because it is based on the Macintosh System Software, which was changed at that time.

[3]For an example of these effects, the reader is encouraged to examine the evolution of Microsoft's Win32 API or some of the core MacOS APIs.

For example, consider a simple service integrated into a framework. Modules depending on that service can begin coding to its interface right away, and a very simple implementation of that interface can be constructed without precluding a fully featured implementation later, which can be added seamlessly as the application matures.

3.3.3 Testing

An obvious benefit to a modular architecture is in testing. Each module should depends only on other modules or services, so replacing each module with a matching test module becomes simple. It becomes possible to test any subset of the system, assuming each module has a test harness counterpart, from single modules to the system as a whole. The component mechanism also precludes the need for recompilation against a test bed when testing.

Chapter 4

The SpecTRM Foundation Classes

This chapter discusses the concrete and abstract aspects of the design of the SpecTRM Toolkit Foundation Classes (STFC) and presents the argument that the design fulfills its requirements. The requirements of the design, simply stated, are that it be maintainable and robust in the face of changing requirements. The framework, or as it is also called, foundation classes, provide basic common services and support structure to all the individual tools in the tool kit.

4.1 Design of the Foundation Classes

The modular framework architecture provide two main sets of features to the application, mechanisms and services. Mechanisms are intended to be integrated into applications using the framework, while services are intended to be used by applications and modules. The foundation classes do not constitute a completed application in and of themselves. They provide much of the skeleton of the application, but the important specifics are deliberately left out in furtherance of the framework's goals.

The implementation of the foundation classes was done in the Java language, and much of the design is presented in reference to that language for convenience. The foundation classes are implemented in Java for a number of reasons, including the fact that Java provides at the language level several complex features that would otherwise have to be implemented for other language environments, and that the Java

class library as provided by Sun Microsystems is very comprehensive. So it might be said that Java was chosen simply because of convenience, but that is not the whole story. Portability is another part of the picture, as well as platform independence. While the goal of "write once, run anywhere"[1] has never actually been met by Java, it is a close enough approximation that careful programming can make up the difference.

4.1.1 Mechanisms

The mechanisms provided by the foundation classes allow the application to be developed in as modular a fashion as possible. The component system is a general requirement of modularity, while the artifact mechanism adds a very flexible system of application data management for use by the application.

The Component Mechanism

The component system allows modules of the system to have consistent interfaces but not be tied to any specific implementation. Using Java as an implementation language in this case greatly simplifies the creation of a component system, as Java provides at the language level several of the features needed. Most important among them is the notion of interfaces, which allow implementations to be divorced from type specifications. The component model used in the STFC is based entirely on Java interfaces in implementing the necessary black box abstraction. Another aspect of a component model, self discovery, is also provided by Java, in the form of its reflection mechanism. Extraneous or experimental capabilities can be integrated into modules and used by client code without altering the module interface, as well as allowing code to use modules abstractly. While the component mechanism does not enforce orthogonality of modules, as indeed it can and should not, it does serve to enable a greater degree of orthogonality.

The component system is defined primarily by a single Java interface, the Component interface. This interface describes the features shared by all components in the

[1] A mnemonic for the claim that Java binaries can be compiled once and run on any computer having a conforming Java Virtual Machine installed.

system, and all components in the system must implement it. Specific module's component interfaces are defined by extending the Component interface. The Component interface defines a set of functionality all components must provide:

- A stream interface that allows components to exchange data in flexible ways without overburdening their interfaces. This is probably the most experimental feature of the component system, and may well prove to have very limited usefulness and need to be removed.
- Identifier accessors for component type and instance, so individual components can be programmatically identified in a consistent way.
- Status accessors so that the status of all components potentially in the system can be ascertained. The system allows for missing and malfunctioning components and can deal with them in constructive ways.
- The component system also requires the overriding of several basic Java mechanisms:
  - Construction and finalization of components are controlled by the framework, so components must define the OnConstruction and OnFinalization methods. By controlling the construction and finalization of components, the framework can handle framework-specific resources devoted to components transparently.
  - Equality testing and the hashCode operation are required to be overridden by all components. The default overridden behavior in the AbstractComponent defines two component instances as being equal if they are the same component, but this definition may be inaccurate or inadequate for some components. The hashCode operation is also required because of the ways in which the Java class libraries depend on the hashCode of equal objects also being equal.

- The conversion to the String type is required of all components. This allows the framework to easily output diagnostic messages without requiring specific functionality of every component.

A number of specific component types are defined by the framework to simplify application development and provide the functionality needed by the artifact mechanism. Other component types have been defined to fill out the functionality of the tool kit.

- **ArtifactHandler** components implement the infrastructure of the Artifact system, providing the core logic for using codecs and associating artifacts with components other than themselves.

- **StorageHandler** components are the core component for all possible interfaces to data storage, defining an interface that accepts universal resource locators (URLs) and produces a Stream that has an associated MIME type.

- **Codec** The codec component serves to convert data of one MIME type to another. It is used in the intermediate process of converting data retrieved from storage into artifact objects in the system.

- **Edit** Edit components provide an interface for manipulating artifacts. In general each Edit component deals with a specific artifact type and none other, but this notion is not enforced, only encouraged for the sake of orthogonality.

- **Simulator** Simulator components execute their input under the control of another component. At present, the only sort of input that is executable is the framework-internal SpecTRM-RL language representation, which implies a single simulator component. However, it is conceivable that simulators with different behaviors or properties could be introduced, as well as simulators that use alternate language forms for the purposes of experimentation.

- **Display** components consist of non-interactive data displays, which are usually static, but may change over time.

- **Interaction** components allow simulation of the user experience in conjunction with simulator components, and can be used to gather user feedback.

- **Analysis** components accept structured data at their input stream, performing some manipulation on that data and outputting the modified data stream on their output stream. Analysis components can easily be strung together to perform powerful operations.

- **Functional** components are primarily event sources for simulator components via their output stream, but can represent any sort of general computation or action that does not operate on data streams. One example might be plug-in components for use by an Edit component.

These components make up the minimal set of component types that are necessary to implement all current and planned features of the tool kit.

The Artifact Mechanism

Artifacts in the tool kit are considered to be any sort of user manipulatable data, from specification documents, projects and compiled simulation code to stored settings and preferences. All concrete data produced by the system is considered to be an artifact, so the definition is very close to the sense in which the word is used in business contracts when referring to deliverables.

The artifact mechanism allows multiple types of data to be manipulated by the tool kit in an orthogonal way. Each type of artifact has an associated handler encapsulating all the logic for using that artifact, including all the knowledge about which other components can be used in conjunction with it.

Each artifact type extends the Artifact interface to form a new interface that describes how to interact with that sort of artifact. The base Artifact interface designates means of tracking artifact dependencies and dealing with composite artifacts, defined as artifacts which themselves contain artifacts, such as a project artifact that contains document and specification artifact types.

The ArtifactManager singleton serves as a central point of contact for the system, and keeps a collection of artifacts currently in the system for short-circuiting artifact and URL requests. The ArtifactManager also does searches in conjunction with the ComponentManager for ArtifactHandlers to handle specific artifacts. The purpose of this feature is to encapsulate the matching process in anticipation of needing to change the algorithm at some future date.

An example scenario of how the artifact mechanism actually interacts with the rest of the system can be found in Section 4.2.2.

4.1.2 Services

Services in the the foundation classes are conceptually distinct from mechanisms in that they are meant to be used by the tool kit and not integrated into the tools. They are also intended to be self contained and have static interfaces.

The Settings Service

The Settings Service manages a hierarchical chain of settings for the application allowing defaults to be derived from any parent level and variable numbers of modules operating on any given level. The abstraction provided is similar to an environment in any block-structured language.

The Settings Service consists of a settings manager that stores settings service configuration, and the settings object, which handles the bookkeeping of looking up values in the hierarchical tree. Settings objects are meant to be serialized with the containing object and only rely on the manager to determine who their enclosing scope should be and when they should promote values to that scope.

The SettingsManager class is a singleton that stores information about how different types of settings are related and manages storage and retrieval for singleton settings types, which generally represent system-wide default.

The Settings class simulates an environment from which values can be retrieved. It optionally has an enclosing settings environment from which it can fetch values

that are exposed but not shadowed. It has a set of exposed values, and a mapping for values it actually defines. It can be serialized as part of any class that uses it.

The Logging Service

The Logging service acts as a point of contact for the rest of the application, providing logging facilities. Logs can be prioritized, sorted, collated, and filtered. Current logs can also be reviewed by the application as it runs, for debugging purposes.

The logging service consists of a Logging Manager, which sets up logs, sets up priority levels, and manages log settings. Logs can be addressed by name, identifier, object reference, or by doing a lookup on a target message. Logs can be buffered for performance or unbuffered for diagnostic purposes. Unbuffered logs are guaranteed (within the limits of the Java runtime system) to be up to date with every message written. Logs can be directed to any sort of stream, file or otherwise. Messages are sent to specific logs based on priority and origin.

The Logging Manager class is a singleton that stores information about the various logs, priority levels, and dispatch rules. The Logging Manager uses the Settings Service to maintain its settings information across system shutdown.

This class implements the stream interface, and wraps a stream of the appropriate type. It also performs buffering and can automatically set up streams at the request of the Logging Manager.

The Message Service

The message service provides message substitution for internationalization purposes. Messages to be retrieved can be based on strings, constants or other identifiers. Messages are substituted for other messages according to a user installed mapping. Messages and metadata can be retrieved from settings, property files, archive manifests or streams. Message service objects can be attached to different classes for ease of use and flexibility.

The Exception Service

The exception service provides a simple means for objects in the system to raise meaningful exceptions without duplicating code or writing complicated code to construct meaningful messages. The exception service is meant to be tailored slightly by each class using it. It can accept a set of strings or objects implementing the runnable interface that produce strings and indicators about where arguments to each exception message creation are to be inserted in the list. Typical usage would be for a class to have an instance of the ExceptionService class and a static method that calls it appropriately for use by the rest of the object's methods.

4.2 Notes

These notes provide additional material relevant to the design, but not essential to understanding the overall structure and intent. They include illustrative examples, examinations of common usage scenarios, and discussions of rationale.

4.2.1 Implementation Notes

Concurrency

Service classes must be thread safe and reentrant if possible. Certain conditions may cause methods in those classes to block, but this should be the exception to the rule. The logging service in particular must be able to handle many messages from several threads simultaneously without having a noticeable impact on the performance of the application. Services in general should not incur any penalties on the application unless absolutely necessary.

Component implementations should also be thread safe. Robustness in the face of concurrency is a must in a heavily thread-based environment like the Java runtime system.

Declaring and Using Exceptions

Each package in the system should have a single top-level exception class used by all functions in that package to raise exceptions specific to classes in the package. This prohibition comes about because the dispatch mechanism in Java deals oddly with exceptions when it comes to signature matching. Also, catch clauses become very unwieldly when more than two or three different exceptions need to be dealt with explicitly. It is much easier to simply declare the raised exception as the root exception type for that package and raise some derived type if necessary. The client can sort out what actually happened, but more often than not, exceptional conditions are not dealt with very gracefully anyway, so it does not matter.

4.2.2 Scenario Documentation

Opening an Artifact

- A request to visit a URL is generated, and sent to the Artifact Manager.
- If the artifact is present in the system, the appropriate Artifact Handler is invoked to service the request.
- If not, the Artifact Manager invokes the Storage Manager to bring the Artifact into the system.
  - A request to locate a URL is received by the Storage Manager.
  - The Storage Manager queries the Component Manager for appropriate components to handle the request.
    * The appropriate Storage Handler Component creates a stream from a URL, and deduces the MIME type of the stream.
    * A Codec Component converts the stream into a Java Artifact Object.
    * The artifact is delivered to the Artifact Manager.
- The artifact manager searches the available artifact handlers for one that can handle this particular artifact, and sends it the Artifact, the URL, and the

MIME type, and registers the Artifact as being present in the system.

  - The chosen artifact handler locates (through the component manager) the necessary components it requires to present the artifact to the user, and instantiates them if necessary.
  - Sometimes, the artifact handler will be the same one that deals with whatever user interface element generated the original request, in which case, everything works out.

- If no Artifact Handler is found, the artifact manager should raise an exception.

Storing an Artifact

- A request to store an artifact is generated and passed to the artifact manager.
- The artifact manager finds the associated URL and queries the Storage Manager for its status.
- The storage manager locates the appropriate store object or instantiates one if necessary.
- The storage manager takes the artifact and uses codecs to convert it into the desired format.
- The storage manager passes the data to the appropriate store for storage.

4.2.3 Illustrative Examples

Use of Universal Resource Locators in the Toolkit

Universal Resource Locators (URLs) are used extensively in the toolkit to link artifacts to one another and even between different portions of an artifact. More information on the structure and meaning of URLs in general is available from RFC 1738. In this respect, all artifacts support hyper-linking. In general hyperlinking is used to convey dependencies to the user and to allow specifications to be self referential.

In practice, this works by using only URLs to identify artifacts not in the system, and for storage purposes artifacts in the system must store their origin anyway, so they too can be consistently referred to by URLs. Artifacts that have been loaded into the system can be referred to by the artifact URL pseudo-protocol as well.

Level one, two and three intent specifications use references heavily, and the use of URLs reflects this.

4.3 Goals

The goals of this design were to produce a maintainable, extensible and flexible foundation for the SpecTRM tool kit. These goals have been met. As has been described in the previous chapter and this one, the design is a modular framework that adequately supports all the necessary functionality and is both maintainable and extensible.

4.4 Future Needs

The design is sufficient to meet future needs for two reasons. The first is that the modules are generally orthogonal to each other — they do not have overlapping functionality or responsibilities. The second is that the framework is extensible, so it need not anticipate future needs, only allow for their existence and uncertain nature.

The elements of the framework are orthogonal because they either break up complex responsibilities and functionality into smaller pieces that are themselves orthogonal, in the case of the Artifact mechanism, or they encapsulate them instead, as in the case of the Edit component. The individual components of the artifact mechanism exhibit orthogonality in that they each have one or two well defined (in terms of interface and specification) responsibilities and no class shares responsibility for performing a given task. Any class used in implementing the mechanism is therefore transparently replaceable at any point as long as its responsibilities do not change. It may seem that the mechanism does not exhibit orthogonality as many of the classes depend on others for certain functionality. Orthogonality is not violated in these cases, because

there is a clear chain of command and each class involved has a specific responsibility that involves no ambiguities. Thus the responsibility of the controlling object is to coordinate the objects doing the work. Orthogonality is possible only when black box abstractions are used and when units are functionally distinct. The mistake made by many object-oriented designs with respect to orthogonality is omitting the second criterion. Black box abstraction is relatively easy to obtain, if only by publishing only specifications and carefully constructing them to be explicit in what is not guaranteed as well as what is guaranteed. Orthogonality, on the other hand, is a much more difficult proposition. The tendency in growing systems is to cluster functionality to close too data, because it is easier to write such code[3]. The result becomes a small number of or even a single class, which rapidly becomes a maintenance nightmare. Even refactoring, while it can be used to break up such a behemoth, cannot automatically isolate distinct functionality. So, while orthogonality is a laudable goal, very little seems to be known about how designers can achieve it.

Extensibility is usually gained at the expense of orthogonality. Generalizations of classes to form hierarchies are brittle, and the shared responsibility present in the parent class makes it non-orthogonal to its derived classes. In a component system, inheritance does not generally occur, only responsibility chains[2]. Extensibility comes mostly from maintainability, orthogonality and encapsulation of policy. Orthogonality has already been discussed, as has maintainability, which arises mostly from properties of the code itself and not the design. The only remaining aspect is policy. Policy refers to the situation in which an object enforces behavior on its clients. An example of this might be a set abstraction that forces its clients to refer to members of the set according to their position in its representation. While this may be a very contrived example, it illustrates how easily abstractions can accidentally enforce policy because of their implementation. In more useful cases, it is possible to design interfaces to such abstractions that prevent them from enforcing policy. In practice, this usually means that the interfaces are as simple as is feasible, because introducing other factors into the interface that do not relate most directly to the abstraction at hand tends to encourage the implementors of those objects to enact policy. The interface for each

component in the framework was designed with this goal in mind, but only time can tell if this aspect of the design will be successful.

4.5 New Development

New development with the tool kit can take one of two forms: new components can be added to introduce new functionality, or new component interfaces can be added to refine or extend existing interfaces or create wholly new component types.

To view the problem from a top-down perspective, new development stems from new needs not covered by the current code base. New functionality will either be similar enough to current functionality, adding new output formats via codec components, for instance, or it will require new component types. Either way, existing components, due to the orthogonality of their interfaces and the black box abstraction using a Java interface gives, will be unaffected by introduction of new components or component types. The only possible exceptions are manager classes which provide convenience operations for dealing with components.

Appendix A

Framework Interfaces

```
package spectrm.framework.component;

/**************************************************************************
 * Interface: Component
 * Author: Patrick Anderson
 *
 * Notes: The Component interface defines methods common to all
 *        framework components.  Every component should implement this
 *        interface or an interface which extends this one.
 *
 * Last Modified: 05/16/2000
 **************************************************************************/

interface Component {

  /**
   * OnConstruction and OnFinalization provide a means for components
   * to have initialization and finalization code while still turning
   * over control of the construction and finalization process to the
   * framework.
   */
  public void OnConstruction();
  public void OnFinalization();

  /**
   * All components must provide overridden versions of certian operations.
   * Each operator is present for a very specific and usually technical
   * reason.
   */
  public boolean equals( Object rhs );
```

```
public int hashCode();
public String toString();

/**
 * These methods define the stream interface for components.  Since Java
 * streams consist of Objects, this is a perfectly general mechanism.
 * The interface is intended to allow components to exchange data without
 * burdening their interfaces with explicit methods.
 *
 * The StreamType class is a type which encapsulates an enumeration that
 * has the standard values of Input, Output, Error, Exception, and Data.
 *
 * Unfortunately the Stream argument must be of type object as
 * the InputStream and OutputStream classes have no other common ancestor.
 */
public Object getStream( StreamType type );
public void setStream( StreamType type, Object stream )
  throws ComponentException;
public void addStream( StreamType type, Object stream )
  throws ComponentException;
public Object removeStream( StreamType type, Object stream )
  throws ComponentException;

/**
 * The following eight methods are convienence methods included
 * for the benefit of readability.  The overhead is considered minimal,
 * as they are trivially implemented in the AbstractComponent class.
 */
public InputStream getInputStream();
public OutputStream getOutputStream();
```

```
public OutputStream getErrorStream();
public OutputStream getExceptionStream();

public void setInputStream( InputStream stream )
  throws ComponentException;
public void setOutputStream( OutputStream stream )
  throws ComponentException;
public void setErrorStream( OutputStream stream )
  throws ComponentException;
public void setExceptionStream( OutputStream stream )
  throws ComponentException;

/**
 * The isNull operation allows for the creation of a Null object pattern
 * for Components.  This allows the null subclass to fail gracefully and
 * removes much redundant reference checking.  Specific implementors of
 * the Component interface should provide factory methods to their clients.
 */
public boolean isNull();

/**
 * ComponentIdentifier is part of an Identifier pattern which uniquely
 * differentiates all component types.  The pattern allows for changing
 * identifiers and schemas for differentiation.
 */
public static ComponentIdentifier getComponentIdentifier();

/**
 * InstanceIdentifiers identify specific instances of a component in the
 * same manner as ComponentIdentifiers.
```

```
    */
   public InstanceIdentifier getInstanceIdentifier();

   /**
    * The ComponentStatus is another enumeration class which describes the
    * overall status of the component.  Possible values include: Functional,
    * NonFunctional, Missing, and Unknown.
    */
   public ComponentStatus getComponentStatus();

   /**
    * getStatusInfo allows components to implement component specific
    * diagnostic and debugging status.  The actual result of this method
    * is component specific.
    */
   public StatusInfo getStatusInfo();

}
```

```
package spectrm.framework.component;
import spectrm.framework.artifact.Artifact;
import java.net.URL;

/**************************************************************************
 * Interface: ArtifactHandler
 * Author: Patrick Anderson
 *
 * Notes: The ArtifactHandler component interface specifies the basic
 *        operations common to all artifact handlers throughout the
 *        mechanism.
 *
 * Last Modified: 05/16/2000
 **************************************************************************/

interface ArtifactHandler extends Component {

  /**
   * Tests for whether or not this handler can cope with specific artifact
   * types.  While it is not yet the case that a single handler and
   * associated components can deal with multiple artifact types, it
   * is possible that integrated editors for compound artifacts may arise.
   * This interface decouples the matching process from the ArtifactManager.
   */
  public boolean supportedArtifact( Artifact artifact );
  public Set getSupportedArtifacts();

  /**
   * Directs the handler to take stewardship of the artifact.  Usually,
```

```
     * this will mean organizing other components to actually handle the
     * editing or display of the artifact.
     */
    public boolean handleArtifact( Artifact artifact );

    /**
     * This alternate form which takes a URL is a convienence construction.
     * The implementation simply calls on the StorageManager to produce
     * the named artifact and then behaves exactly as the previous method.
     */
    public boolean handleArtifact( URL url );

}
```

```
package spectrm.framework.component;
import spectrm.framework.artifact.Artifact;
import java.net.URL;

/**************************************************************************
 * Interface: StorageHandler
 * Author: Patrick Anderson
 *
 * Notes: The StorageHandler component interface specifies the basic
 *        interface common to all StorageHandlers.  The StorageHandler
 *        encapsulates the process of dealing with artifacts in relation
 *        to persistant store.  Encoding and decoding is done by using
 *        Codec components.
 *
 * Last Modified: 05/16/2000
 **************************************************************************/

public interface StorageHandler extends Component {

  /**
   * Some protocols require persistent state in order to operate correctly.
   * This call allows such protocols a means of estabishing that state.
   */
  public void associate( URL url )
    throws ComponentException;

  /**
   * Establishes whether the given protocol and location are supported by
   * this handler.
```

```
   */
  public boolean supportedURL( URL url )
    throws ComponentException;

  /**
   * Retrieve an artifact from the associated store.  Raises an exception
   * if the URL is not meaningful or is not associated with this store.
   */
  public Artifact getArtifact( URL url )
    throws ComponentException;

  /**
   * getAvailableArtifacts returns a set containing the URLs of all the
   * artifacts in the associated store.  If this StorageHandler requires
   * setup by way of the associate call, then the set will be null if the
   * given URL doesn't have enough in common with the associated URL.
   */
  public Set getAvailableArtifacts( URL url );

  /**
   * Stores an artifact in the associated storage location.
   */
  public void putArtifact( Artifact artifact, URL url )
    throws ComponentException;
}
```

```
package spectrm.framework.component;

/**************************************************************************
 * Interface: Codec
 * Author: Patrick Anderson
 *
 * Notes: The Codec component interface defines a component which
 *        translates data from one form to another.  Codecs can be
 *        unidirectional or bidirectional, and should support both
 *        the streams interface and explicit operation using encode
 *        and decode method calls.
 *
 * Last Modified: 05/16/2000
 **************************************************************************/

public interface Codec extends Component {

  /**
   * Describes the type of data this codec accepts, using framework
   * specific MIME types (RFC 1521).
   */
  public MimeType getInputType();

  /**
   * This method should almost always return the type associated with
   * framework artifacts, but might not always do so, for instance in
   * the case of translation between framework internal data
   * representations.
   */
```

```
  public MimeType getOutputType();

  /**
   * An explicit encoding operation on the target object which returns
   * an appropriate object representing or containing the result.  The
   * target object must represent the input type of the codec.
   */
  public Object encode( Object obj );

   /**
    * Identical to encode, except that the target object must represent
    * the output type of the codec and the operation may throw an exception
    * if the codec is unidirectional.
    */
  public Object decode( Object obj )
    throws ComponentException;
}
```

```
package spectrm.framework.component;
import spectrm.framework.artifact.Artifact;
import java.net.URL;

/**************************************************************************
 * Interface: Edit
 * Author: Patrick Anderson
 *
 * Notes: The Edit component interface defines the interface for both
 *        data manipulation and read-only data display components.
 *
 * Last Modified: 05/16/2000
 **************************************************************************/

public interface Edit extends Component {

  /**
   * Opens the given artifact for editing and display.
   */
  public void open( Artifact artifact );

  /**
   * Repositions the current editing position within the artifact to
   * the location specified by the given URL.  URLs must either be
   * artifact specific or have locality with the artifacts which are
   * currently open in the Edit component.
   */
  public void reposition( URL url );
```

```
  /**
   * Returns the URL representing the location closest to the current
   * editing position.  The structure of the URL is heavily artifact
   * dependent and should be used as a magic cookie whenever possible.
   */
  public URL getCurrentPosition();

  /**
   * Roughly equivalent to save and save as.  Provided for access by the
   * reflection mechanism for programmatic control of the component.
   */
  public void store( Artifact artifact );
  public void store( Artifact artifact, URL url );

  /**
   * Discard any changes to the artifact and release it from this component.
   */
  public void close( Artifact artifact );

  /**
   * Returns a set containing the artifacts currently in use by this
   * component.
   */
  public Set openArtifacts();
}
```

```
package spectrm.framework.component;
import spectrm.framework.artifact.Artifact;

/**************************************************************************
 * Interface: Simulator
 * Author: Patrick Anderson
 *
 * Notes: The Simulator component interface defines the interface for
 *        all state machine based simulation components.  Incoming
 *        events arrive on the component's input stream, and outgoing
 *        events are pushed into the component's output stream.
 *
 * Last Modified: 05/16/2000
 **************************************************************************/

public interface Simulator extends Component {

  /**
   * Sets up a simulation of the specified artifact, which should
   * represent data which is meaningful to the simulator.
   */
  public void simulate( Artifact artifact );

  /**
   * Allows the simulator to accept initial conditions for the simulation.
   */
  public void initialize( Artifact artifact );

  /**
```

```
    * Starts the simulation.
    */
   public void start();

   /**
    * Runs the simulation through a single step.
    */
   public void step();

   /**
    * Stops the simulation.
    */
   public void stop();
}
```

```
package spectrm.framework.component;
import spectrm.framework.artifact.Artifact;

/**********************************************************************
 * Interface: Display
 * Author: Patrick Anderson
 *
 * Notes: The Display component interface defines the interface for all
 *        components which display static or dynamic data to the user.
 *        Dynamic displays should recieve updates via the streams
 *        interface.
 *
 * Last Modified: 05/16/2000
 **********************************************************************/

public interface Display extends Component {

  /**
   * Provide data to the component for display.
   */
  public void initialize( Artifact data );

  /**
   * Direct the component to focus the display on some portion of the data.
   */
  public void focus( URL position );
}
```

```
package spectrm.framework.component;
import spectrm.framework.artifact.Artifact;

/**************************************************************************
 * Interface: Interaction
 * Author: Patrick Anderson
 *
 * Notes: The Interaction component interface defines the interface for
 *        all components which provide user interaction with simulations
 *        or other components.  Interaction components should cooperate
 *        with other components mainly through the streams interface.
 *
 * Last Modified: 05/16/2000
 **************************************************************************/

public interface Interaction extends Component {

  /**
   * Sets up the display and interaction aspects of the component.
   */
  public void setup( Artifact description );

  /**
   * Allows the component to have any internal state set.
   */
  public void initialize( Artifact state );

  /**
   * Produces a copy of the current internal state of the component.
```

```
    */
    public Artifact dump();

    /**
     * Freezes the component at the point the method was called.  Further
     * input may be discarded or queued depending on the implementation.
     */
    public void freeze();

    /**
     * Causes the component to resume responding to user and stream input.
     * If input was buffered, all incoming events are processed immediately
     * to make the state consistent.
     */
    public void resume();
}
```

```
package spectrm.framework.component;
import spectrm.framework.artifact.Artifact;

/**********************************************************************
 * Interface: Analysis
 * Author: Patrick Anderson
 *
 * Notes: The Analysis component interface defines the interface for
 *        data analysis components.  Incoming data arrives on the input
 *        stream and processed data is pushed out the output stream,
 *        allowing for chaining of analyses.
 *
 * Last Modified: 05/16/2000
 **********************************************************************/

interface Analysis extends Component {

    /**
     * Initialization provides a means for the component to accept any
     * initial state needed to perform it's analysis.
     */
    public void initialize( Artifact state );

}
```

```
package spectrm.framework.component;
import spectrm.framework.artifact.Artifact;

/**************************************************************************
 * Interface: Functional
 * Author: Patrick Anderson
 *
 * Notes: The Functional component interface defines the interface for
 *        components which perform some work in the system, for example
 *        providing input events to the simulator.
 *
 * Last Modified: 05/16/2000
 **************************************************************************/

interface Functional extends Component {

  /**
   * Allows the component to accept initial state.
   */
  public void initialize( Artifact state );

  /**
   * Since the functional components are not generally stream based,
   * this method allows the user of the component to prod it into
   * performing whatever work it does.
   */
  public void activate();
}
```

Bibliography

[1] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000.

[2] Erich Gamma ... [et al.]. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[3] William J. Brown ... [et al.]. *AntiPatterns.* John Wiley and Sons, Inc., 1998.

[4] Brian Foote. *Designing to Facilitate Change with Object-Oriented Frameworks.* Masters thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1988.

[5] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[6] Jr. Frederick P. Brooks. *The Mythical Man-Month.* Addison-Wesley, 1995.

[7] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley, 2000.

[8] Nancy G. Leveson. Completeness in formal specification language design for process-control systems.

[9] Nancy G. Leveson. Sample intent specification: Altitude switch.

[10] Nancy G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[11] Nancy G. Leveson. Draft intent specifications (including spectrm-rl) user manual. 1999.