GARDEN OF APPLICATIONS

INTRODUCTION

This pattern language's initial growth and development was borne out of two forces. The first was that I had completed a challenging development effort in Java which had some promising ideas for a pattern language. The second was that during the project, I had the great good fortune to read Christopher Alexander's *The Timeless Way of Building* and later, *A Pattern Language*.[1]

The System

Our system delivers real-time security prices and news headlines to users via a Java-enabled web browser. My team was given the task of establishing a framework which would launch these and other market data-related applications. It was an interesting project because one of the design criteria given to our team by the users was, "It is not clear what the final system was going to look like, so the system had to be easily changeable." Our initial reaction was, "How are we ever going to build a framework if we can't establish the baseline functionality?" After some initial attempts at writing large-scale architectural documents that were largely thrown back to us, we put our heads together and designed a set of simple constructs. These included *Backplane*, *Configuration Services*, *Standard Application*, and *Desktop*. We then proposed a standard decomposition of the applications into *Service*, *Business Object*, and *Application* – this latter component being similar in spirit to the View in a MVC architecture.[2] From the standpoint of the framework, the core component was the Backplane. The only components it knew about were the Configuration Service, Standard Applications and the Desktop.

The backplane was the core of the system and was derived from the Java applet class (java.awt.applet). As readers who have worked with this class already know, applets have a strong notion of lifecycle with methods which are called by the browser and correspond to the states **init**[sic], **start**, **stop**, and **destroy**. After some consultation with the developers and some introspection on my part, I saw that what was needed was the chaining of these state changes to the applications which were launched via the backplane. Additionally, we took advantage of Java's late binding mechanism which allows programmatic loading of classes into the Java Virtual Machine and "elevated" this construct to the level of loading entire applications.

Not long after other developers had started writing applications, we were given the request to change some of the core functionality which affected the way that news stories were to be displayed. At this point, the *Interface Repository* came into being, inspired by a similar construct in CORBA. A variation on the standard applications could be automatically started by the backplane at start up so it seemed like a logical extension to use these "special" standard applications to populate the interface repository with implementations of standard facilities that other applications could use as part of their normal operation. This allowed us to neatly define (and subsequently redefine) core services without having to modify the backplane itself. This became the basis of the *Bootstrap Application* described herein.

The last change that was made to the backplane was the addition of some convenience functions for instructing a given application that a new view (most applications supported multiple views) needed to be created. As part of view creation, the caller could pass in an application-specific argument (i.e., a

---

[1] The importance and relevance of Alexander's work in the software design patterns community is well established in the literature. For the interested reader, an excellent introduction is at http://gee.cs.oswego.edu/dl/ca/ca/ca.html (Doug Lea's article on Christopher Alexander).

[2] Actually, even this decomposition, although standard was driven by a desire for changeability. In particular, the Service layer often represented the layer in which network connections and data were sent received through. However, since it was not clear what the ultimate transport mechanism was going to be, we knew that our design had to work with "anything."

java.lang.Object). This was much in the spirit of the Memento pattern in the book, *Design Patterns* by Gamma, et.al.

Once the basic constructs were in place, adding new functionality to the system became a very straightforward exercise. In discussing these ideas with other developers, it has become clear that this is a very flexible and easy to use set of constructs.

This pattern language is focused primarily on constructs that were required for end-user applications. In particular, this system provides a suite of inter-related applications. I have not tried these specific patterns in the context of server-side functionality, however some of these constructs can be seen in toolkits such as the recently released Java Server Toolkit. This seems to suggest that certain idioms in Java are universal, independent of whether the application in question is an user application or is server-based.

Patterns

In conversations with other developers, both on and off this particular project (even ones who had not been "enlightened" by the concept of design patterns), it became apparent that these constructs were intuitively understandable. It was easy to discuss the system by simply using these terms. The next logical step was to write a pattern language based on these constructs. This document represents the beginnings of that effort. It is inspired strongly by Alexander's notions of piecemeal growth as discussed in *The Timeless Way Of Building*. Of particular interest was the section entitled, The Way, and its discussions of constructing and repairing buildings in a piecemeal fashion. Alexander's writings coupled with similar such writings from the patterns community served to catalyze the realization that, "Yes, it is possible to do system development without a roadmap carved in stone."

I was also greatly struck by James Coplien's essay in *The C++ Report*, "Space: The Final Frontier." This work crystallized a number of concepts, particularly the notion of the runtime architecture of a system. I found this to be very interesting because one of the core components of the system we had developed, the interface repository, was almost a trivially simple construct. Yet at runtime it took on a life of its own. When coupled with the dynamic application loading (another purely runtime construct) the interface repository became a very powerful mechanism for adding functionality to the system.

This Pattern Language

Readers of Alexander will see that these patterns are written in the style of *A Pattern Language*. I am the first to admit that these are crude in comparison to his – to that, all I can say is, let's see how they look in another ten years! This language is also incomplete – the key constructs are there, but there were other constructs which contributed to the success of the system, these need to be documented too. To help illustrate some of these concepts more concretely, I have included an implementation in Java of several of the constructs. See Appendix A for more detail.

Acknowledgements

This paper is dedicated to my wife, Eriko Harukawa. Without her inspiration and constant support this paper would still be riding around in my head. I also wish to thank the Sun Java Center at Sun Microsystems in their support for this paper.

Last, but not least, my shepherd, Kent Beck, deserves tremendous credit for pointing the way.

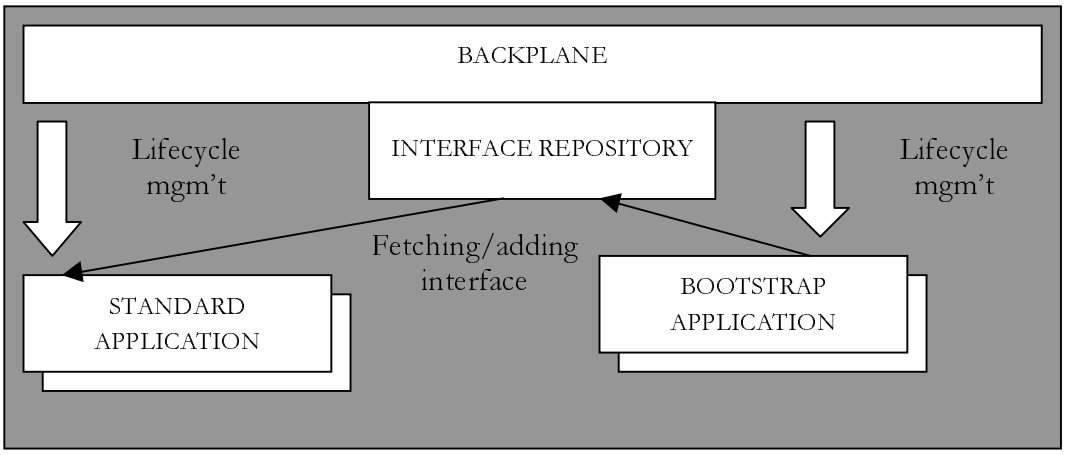
GARDEN OF APPLICATIONS(1)

...systems do not necessarily require master plans in order for development to begin. With the proper meta-constructs in place, their development can proceed in an almost "organic" fashion. This pattern describes such constructs.

We want to build systems which can be enhanced and modified in a piecemeal fashion. This is necessary because the requirements for systems inevitably change, sometimes even before they are deployed for the very first time.

All developers want to build systems which are easy to modify, easy to enhance, easy to debug even in the face of rapidly changing requirements, easy to integrate new technologies, and to allow multiple developers to work on the same system. This pattern identifies some important constructs which address these forces. Here we see a simple schematic of some of the components and their relationship to one another.

As the name of this pattern suggests, we need to provide a set of constructs for the system which can provide a basis for easily adding and modifying applications; an environment in which these



applications can be modified according to their own requirements without unduly affecting unrelated applications. The exact nature of these constructs can vary from system to system, but in each case they provide "behind-the-scenes" support for the applications. Examples include:

1. Notion of standard and bootstrap applications.
2. Explicit constructs to manage the startup and shutdown of standard applications.
3. Window (or view) creation for those applications that require it.
4. Interface repository support.

These and other facilities are discussed in more detail in the subsequent sections of this pattern language. Before we continue with these facilities, we provide motivation for them.

A common misconception about system development is that a detailed "ultra-complete" plan must be in place before the first line of code can be written. This misconception persists even though there is 1) tremendous anecdotal experience suggesting that this approach does not work and 2) there are fundamental reasons for why these so-called master plans do not work is the strongest argument against master plans is that the creation of these master plans is such a lengthy process that the users of the system have already changed the very business practices that were the basis for the plan in the first place. This creates a strong need for an approach to application development which does not require a "down to the last detail" plan to be in place in order for development to begin. An approach which is amenable to frequent change and modification as the requirements of the system and its component

applications change.[3] We especially want to avoid writing throw-away code, i.e., the so-called prototype system that inevitably is discarded when the "real development" begins.

There is a countervailing development philosophy called *piecemeal growth*. This concept, as it is discussed here, is based in Christopher Alexander's book, *The Timeless Way of Building*. Alexander discusses it in the context of creating and repairing buildings and towns. He points out that no building is ever really complete – the inhabitants' requirements change, the environment in which the building exists changes, etc. He also proposes that the meaning we commonly associate with "repair" needs to lose its sense of "incorrectness" and instead view repair as an opportunity to redesign and invigorate the structure in question. This is very similar to the notion of maintenance in the software development community. In the same work, Alexander asserts that the processes required for constructing buildings and even towns must be "generational," because, like living things, buildings and towns must be able to respond to local differences in terrain, material, etc. and of course, the requirements of the inhabitants.

Note, that this is not to suggest that the development of a system should be done in a haphazard way. Instead, it is important that a development process be put in place that makes it possible to develop the system incrementally or to make changes in response to the users' changing needs without undue disruption of the components which do satisfy their current requirements. In fact, it is important that frequent checkpoints and reviews with the users of the system be planned and accounted for in the timeline. Since, this frequent meeting with the users of the system almost guarantees there will be changes, it is important that a foundation which accommodates change be put in place. By using the constructs described in this paper, a basis for piecemeal growth is established. In particular, these constructs provide a kind of software trellis for the rest of the application, i.e., once established they should remain relatively constant. However, like the trellis which supports flowering vines, these constructs should play primarily a support role in the system.

Thus, by deploying the constructs described herein, we provide a means by which changes to existing applications or new applications can easily be easily without unduly impacting other unrelated applications in the system. The most important of these constructs is one called the BACKPLANE(2). This part should be relatively constant, particularly because it really is a meta-construct, i.e., it is a part of the system which is largely outside of the user's application domain. This simplifies its maintenance because we, the developers of the system, are responsible for defining it and any subsequent changes. The application-specific functionality is moved into components called STANDARD APPLICATIONS(3) and optionally, BOOTSTRAP APPLICATIONS(5). Using these constructs helps the developers to provide the required functionality without increasing the brittleness and complexity of the core components, e.g. the BACKPLANE(2). We encourage using this application metaphor, particularly the bootstrap application in conjunction with an interface repository to provide re-useable and/or generic functionality. This helps to reduce coupling between components.

Therefore:

Create a framework onto which the application level functions can easily be deployed. This is done by splitting the entire application (the user's application) into two broad categories, the backplane and standard applications. Make sure to provide facilities, such as an interface repository, to reduce coupling between applications.

To encourage piecemeal growth, we must provide a standard starting place for the application-specific components of the system. This can be done by using a BACKPLANE(2) and STANDARD APPLICATIONS(3). The liveness of the system can be increased via the use of an INTERFACE REPOSITORY(4) since this construct helps to capture the dynamic nature of applications. It also helps reduce the coupling between applications and the framework itself.

---

[3] Please note that I do not attempt to argue this point here, but hope that the reader's own experience with system development has resulted in a similar conclusion (with regards to the applicability of "master plans").

Finally, none of this will work if a REVISION CONTROL SYSTEM has not been implemented. In order for piecemeal growth to work, there has to be a sense (among the developers) that experimenting with the system is “okay” and that there is a buffer between the version in development and the version being used by the users.

BACKPLANE(2)

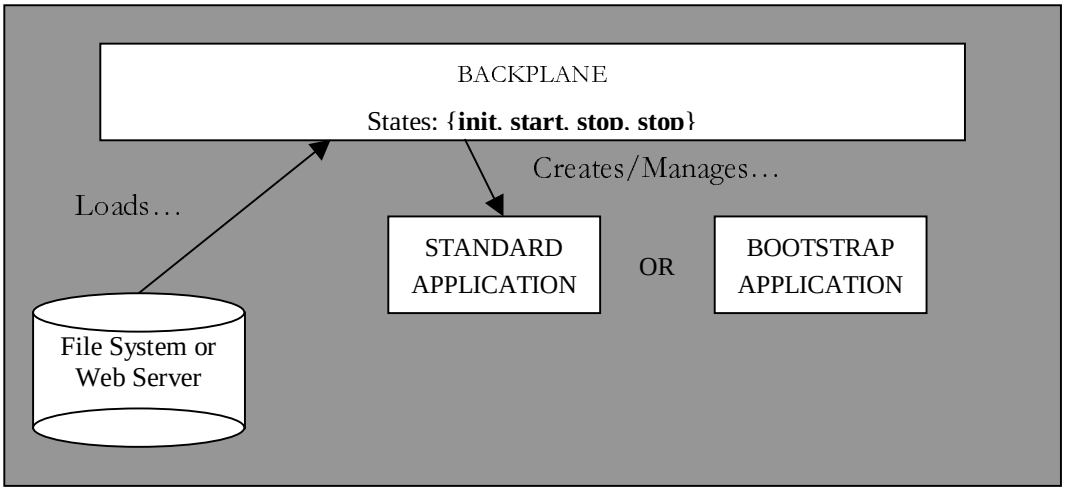
...in building a system according to the ideas introduced in GARDEN OF APPLICATIONS(1), it was necessary that certain parts of the system not be in constant flux. In particular, we identify one core component as a meta-construct against which the applications can be developed and executed

❖❖❖

One core component in the system must act as a meta-construct from which to deploy the user's application. This component must be able to launch and maintain applications as well as provide support for common high-level operations for the applications.

Piecemeal growth requires that certain parts of the system remain stable. Our goal here is to identify the core component that can be made stable, even in the face of incomplete understanding of what the completed application will be. One of the components whose stability is key is a construct which acts as a *backplane* for the system's applications. The name of this construct is borrowed from computer systems engineering – it connotes the same idea as in that discipline. Basically, the backplane provides a standard means of adding new applications to the system. Like its physical counterpart, it does this by defining a number of standard interfaces that an application must satisfy or provide if it is to run in the system. For example, the backplane define standard protocols for how the application is launched, suspended, and terminated; how new windows are created, etc. We call this broad category of functions *lifecycle management*. Aiding in this lifecycle management function, we have supporting constructs such as the INTERFACE REPOSITORY(4) and CONFIGURATION SERVICES.

The backplane is the first component of the system to be loaded at runtime. One of its primary roles is to bootstrap the user's environment and then subsequently provide a standard way for other applications to be loaded.



The backplane should have explicit constructs that center around the lifecycle of the system and its constituent applications. This part of the pattern focuses on the need to make the life cycle of the backplane and the applications explicit. Why is this? From the user's standpoint, the system's architecture is only defined in terms of how they use and interact with it. The traditional notion of application architecture is similar in spirit to a diagram of a house not yet built is – useful to the architect and builder, but not as useful to the inhabitants of the house or in this case, the users of the system. It is critical that the system architecture provides explicit constructs that will map the application's static compile-time constructs into the corresponding runtime constructs.

By making the lifecycle management facility explicit, we signal strongly to the application developer this focus on the active, rather than the passive elements of the architecture. We can certainly find similar such constructs in other application frameworks. For example, most GUI frameworks support

the notion of a callback for GUI event management. This is an example where the temporal relationship between the components is a key attribute and is defined primarily as a runtime construct (for example, it is possible that the relationship was not even defined at compile-time). Lifecycle management extends this construct to the application level. This construct is particularly important to applications which need to support more than simple fork/exec and then exit semantics. For example, the user's applications many need to provide the ability to temporarily suspend execution of a program, by making these state transitions an explicit construct, we give the programmer the ability to support this requirement in a standard way. When this construct of a standard life cycle is used with languages which support late binding, an additional level of functionality can be achieved. These languages encourage developers to take advantage of the ability to defer the choice of implementation until runtime. Thus, late binding increases the significance of temporal forces on the system's functionality and its corresponding ability to change its functionality at runtime. Again, this facility increases greatly the ability of the system to experience piecemeal growth because it allows for easier deployment of new or altered functionality.

As architect, we want to encourage application developers to focus on the runtime architecture of the system – the assumption here being that a more dynamic system will be more useful since it already lays the groundwork for an environment which will support change. If our system does not correctly reflect the runtime characteristics of the system, it will simply remain a collection of classes – useful perhaps in their own way, but never capturing the richness and usefulness of an system architecture which explicitly describes how it will manage the runtime architecture. This is why the notion of lifecycle is important to my vision of how this construct should be used.

Looking at the counter-example of systems without this notion of explicit lifecycle management, it often becomes necessary to manipulate an application's lifecycle in an ad hoc fashion. The issue that arises in this situation is that given the lack of structure in the application regarding its lifecycle, it frequently becomes awkward to support. By making lifecycle management an explicit and necessary construct, we encourage developers to provide application-specific semantics for lifecycle state changes. We also ensure that there is a focus on lifecycle issues, a critical characteristic of systems whose availability to users must be strictly managed.

Interestingly enough, lifecycle management is a construct that has widespread application. The backplane can take advantage of it as well. The backplane should have its own lifecycle – an explicit construct which is exposed to the applications so that they can monitor it, and if necessary, respond to changes in the backplane's state as well. This is very important, particularly to applications which are long-running – perhaps almost as long running as the backplane itself. In particular, the backplane should have a standard startup and shutdown sequence. As we will discuss in the BOOTSTRAP APPLICATIONS(5) pattern, there is a certain class of applications which need to be loaded after the backplane initializes itself, but before the user is presented with any means to launch applications of their own choosing. The important concept is to recognize that the notion of life cycle exists at many different levels in the system and can be used to effect micro- or macro-level operations.

Steering the role of this construct back to the issue of encouraging piecemeal growth, the backplane should not be overly involved in domain-specific functionality. There are better facilities for extending the functionality of the system. Additionally, since the backplane is core code, we wish to avoid modifying it since any changes can have a global effect. Of course, this assumes that the applications in question are reasonably independent themselves.

Nevertheless, there may be certain facilities which may be useful to put into the backplane. For example, if the system has a GUI, it is useful to control the top level display of the GUI interface. This could include providing standard frames for the applications to be displayed in, standard menu navigation, etc. Another facility that is useful for the backplane to provide and is implicitly assumed to exist in this pattern language is a STANDARD CONFIGURATION SERVICE. The purpose of this service is to provide a common way for all applications to read and write configuration information, both global to the system and specific to the applications themselves. Finally, the interface repository is critical to extending the system as a whole and is discussed later in this pattern language.

Beyond providing generic facilities such as a standard look and feel for GUI components, care must be taken when adding domain-specific constructs to the backplane. We want to avoid making the backplane a complex component since that will make it a component which increasingly has more and more explicit dependencies on it. This, in turn, makes it more and more difficult to modify the backplane. Thus, even if the backplane does define implementations of domain-specific functionality,

the interfaces which define this functionality should still be exported into the INTERFACE REPOSITORY(4) and subsequently imported by applications which wish to.

Therefore:

Create a backplane component in the system. This component should be primarily concerned with providing facilities which address the runtime architecture of the system. These include initialization and termination of applications and high level window management. The backplane is also an ideal place to put the interface repository. Avoid placing application or domain-specific functionality into the backplane and instead use an interface repository to support this activity.

❖❖❖

This construct is designed purely as a meta-facility which simplifies the development of new applications and furthers the metaphor of "feature as application." When creating a system which uses a backplane, be sure to use STANDARD APPLICATIONS(3) and BOOTSTRAP APPLICATIONS(5). These, combined with the dynamic facilities in the INTERFACE REPOSITORY(4) can provide a system which satisfies the liveness required by the user's application while providing the stability and separation of function required for developing applications. As the backplane changes its own state in response to external lifecycle changes, it can alert the standard applications either via direct method calls or use a construct such as a MESSAGE BUS[4].

[4] This pattern has been identified in other texts as publish/subscribe, observable/observer, etc. See *Patterns of System Architecture* for an excellent discussion of this pattern.

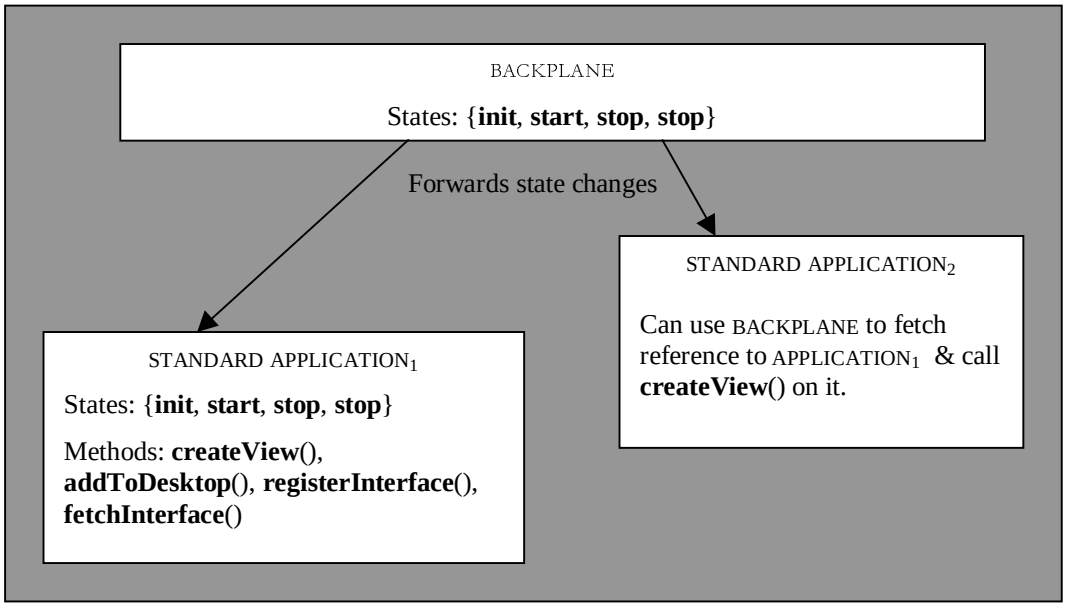
STANDARD APPLICATIONS(3)

...applications should adhere to a structure-giving interface. This allows for explicit manipulation of applications in a standardized way. Following on from the BACKPLANE(2) discussion, the focus on the lifecycle is continued here. Moreover, because this construct encourages modularity, the opportunity for piecemeal growth is increased.

Applications need to have a standard structure so that the backplane may launch many different types of applications. Moreover, the system needs to make clear to developers what facilities their applications must provide in order to run under the auspices of the backplane. Finally, the system needs to provide a standard means of identifying the main entry point for each application.

Each large unit of functionality in a system (the assumption here is that viewing the system as a collection of applications is valid) should have a standard structure from the standpoint of the designer and the developer. In discussing the notion of a "standard application," we assume an OO environment for system implementation. Thus the standard application described herein is typically a super-class of the specific application classes.

For the designer of new applications for the system, this construct puts a stake in the ground by defining what an application is from the standpoint of the backplane. In general, an application will



correspond to an identifiable part of the system, i.e., what a user of the system would consider to be an application or at least an identifiable function that they can perform using the system. Moreover, this construct defines what facilities all application must provide and also what standard facilities are available to an application. For example, a standard application may have a predefined means of adding GUI components to the user's desktop. From the standpoint of encouraging piecemeal growth, this construct is very useful since once it is determined that some new function is useable by all or most of the applications, we can add it to the standard application's definition and thus making it available to all applications.

For the developer of an application, the standard application helps to bootstrap the mental process for many of the same reasons that it is useful to the designer (and in fact, frequently these people are one and the same). In particular, it helps the decomposition from a temporal standpoint. The developers know what assumptions the system makes with regard to loading, initializing, and terminating the application. For example, in some systems, standard applications may be implicitly singletons, in

which the case if multiple views provided by the same application are required the developer will need to implement this capability in the context of a singleton. On the other hand, if the backplane declares a policy which says that there may be multiple instances of the same application, then the developer needs to ensure there are no issues with having multiple instances of application (i.e., such as data corruption caused by improper locking schemes)By making the application an explicit construct, we provide the developer with concrete familiar terrain in which to develop specific applications. All facilities that are defined in the standard application are available to the developer. This helps to ensure consistency across applications. Beyond these typical benefits, the standard application also has indirect benefits. For example, it helps a new developer to come up to speed more quickly because other developers on the project can simply say, "Oh, yeah. Extend STANDARD APPLICATION, update this configuration file and you'll be all set to run and test your application.

Thus, when a developer is given the task of modifying or fixing an existing application, they always have a concrete place to start – just look for the class that implements the abstract STANDARD APPLICATION. It is also interesting to note the parallels between this pattern and Alexander's pattern, MAIN ENTRANCE(110) in *A Pattern Language*. In Alexander's pattern, he discusses the how the main entrance of a building helps to fix its other attributes, and so too, does the STANDARD APPLICATION by defining key operations. Like the entrance way to a building, these features make it easier for a developer, potentially unfamiliar with a particular application, to begin the task of understanding the structure of the application in question. This aids the effort associated with speeding maintenance efforts, reflecting the fact that, even if a developer understands what an application does from a functional standpoint, understanding how that functionality has been implemented is often a non-trivial task. Thus, the more familiar landmarks the maintenance developer has, the quicker they can form a mental picture of the implementation. By making standard application a concrete construct, we encourage the developer to extend this metaphor to the applications that are developed for the system.

When having to understand the temporal structure of an application, i.e., how it responds to lifecycle changes, the task of determining temporal aspects of an existing application is simplified. This task is frequently more complex to ascertain than the static relationships expressed in the code. However, with the lifecycle of the application made explicit, again, it becomes a simpler task to form a mental image of how the application will behave at runtime.

Another important facility to have at one's disposal is a language which supports late binding. This greatly improves the ability to add new applications to the system, even at runtime. It also provides opportunities to reduce the overall footprint of the system, based on the notion that facilities which are not used by the user will not be loaded into memory. This latter issue can frequently be mitigated by the presence of a host operating system which support demand paging. Fortunately, in recent years, robust implementations of languages and environments which support these facilities have become more widespread (for example, Java and C++ with shared libraries).

Therefore:

Make the components with which the user interacts standard applications which provide standard facilities and interfaces. These applications have a standard interface by which the backplane can manage their lifecycle methods. These applications can be dynamically loaded to help reduce the overall footprint of the system. Avoid having applications communicate directly with one another – use the interface repository for this purpose.

A variation on this pattern, discussed later in this document is that of BOOTSTRAP APPLICATIONS(5). Whenever possible, avoid having standard applications communicate directly with one another. Instead, introduce a layer of indirection via the INTERFACE REPOSITORY(4) and have applications exercise each other's functionality via those interfaces.

INTERFACE REPOSITORY(4)

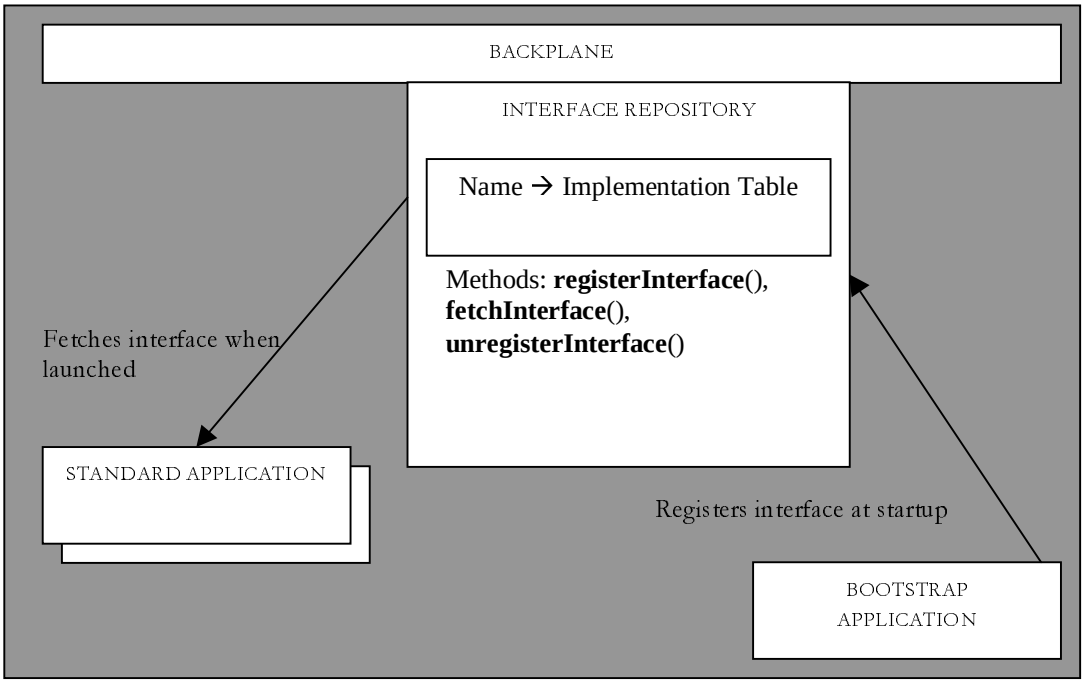
...it is important that we provide a way of adding new functionality to the system without having to modify the BACKPLANE(2) itself. We also want the functionality to reflect the system at runtime and not be defined at compile time or fixed as part of system initialization.

❖❖❖

Applications which need to interact should do so via standard interfaces. Furthermore, as much as possible, applications should avoid referring directly to one another in order to access these facilities. The system should provide a facility which makes this construct easy to use, both from the standpoint of the application exporting a standard interface and for the applications which use that facility.

We need to provide a facility which makes it easier to have applications export their functionality. Moreover, as discussed in BACKPLANE(2), we need to provide a standard facility for extending the functionality of the system without having to add new facilities to the backplane itself. There are a number of reasons for this. The first is statically modifying the system is often not an option. Some systems need to run all or nearly all the time and thus require a more dynamic facility to be deployed. Secondly, changing code, particularly for the purpose of adding new functionality, often has the effect of narrowing the functionality (in the sense of producing further constraints on the domain which it addresses) rather than expanding it and makes the system brittle and potentially more bug prone. The interface repository provides an excellent opportunity for reducing coupling as will be discussed.

Conceptually, the interface repository is simply a mapping between a symbolic name and a public interface associated with which is some well-defined behavior. Applications can then access these



interfaces simply by performing some kind of lookup which either returns the implementation associated with that name or some kind of error value/exception if the name is unknown to the interface repository. The structure of the interface repository may be very simple or it may be very complex. For example, each entry may simply consist of a unique identifier (such as a character string) and a reference to an implementation. Alternatively, it may be very complex such as that provided by CORBA in which there is a hierarchical name space – this is a construct that scales up and down very well. Additionally, this repository needs to provide facilities so that it can change over time. For

example, add, delete, search, and iterate, etc. are operations which need to be provided by the repository. These can be used by applications during their own lifecycle to determine whether a given function is available at a given time. As a variation, the interface repository can be given additional liveness by adding associating real-time events with these operations. This makes it possible for applications to alert other applications as their interfaces become available, are removed, etc.

It is this notion of well-defined behavior that makes this facility very powerful. In particular, the public interface should only provide those functions which are part of the interface's domain – this should be self-evident. However, what makes this facility very appealing, particularly for languages which allow private or protected classes to implement public interfaces (Java and C++ are two such languages) is the ability to expose only the interface's public methods, regardless of how complex the implementation class is. This obviously simplifies life for the developer and/or the maintainer of such code since they can provide a better-defined subset of the required functionality(one hopes!).

Going back to the role that the interface repository plays vis a vis the backplane, this is where functionality can be added to the system without directly altering the backplane. Obviously, any functionality can be added using this facility, thus a reasonable question is what should or should not be put into the interface repository. As a general rule of thumb, the main criteria are whether it augments an existing facility in the backplane or is it a new one. New facilities should remain independent of the backplane. Since this can be a gray area, the rule of thumb is to resist adding new functionality, particularly domain-specific functionality, to the backplane.

Frequently, certain classes of functionality need to be available to many or all applications. We can make it available by loading it into the interface repository with a BOOTSTRAP APPLICATION(5). This allows us to ensure that the functionality will always be available for the users who need it, yet for those who do not, we can avoid wasting system resources unnecessarily (presumably we have a facility for determining what applications a given user needs). Particularly, for languages which allow components to be loaded dynamically, this combination can be particularly effective. Moreover, since only interfaces (i.e., some kind of abstract class) are put into the interface repository, interfaces with multiple definitions can defer their definition of a given interface until they are being loaded.

Therefore:

Create an interface repository for the system and let the backplane serve as the central access point to this list. This list of publicly accessible interfaces should be indexed by a unique name.

BOOTSTRAP APPLICATIONS(5)

...certain functionality needs to always be present in the system yet it does not make sense to add the backplane. This requirement can be satisfied using a variation of the STANDARD APPLICATIONS(3) pattern called a bootstrap application.

There should be a way of ensuring that certain functions are always available to the user's applications. Yet, we want to make sure that these functions are independent of the BACKPLANE(2) and STANDARD APPLICATIONS(3) constructs so that the backplane can either be used in other contexts or we can provide different facilities to different users of the system.

In the course of developing a large system, it is generally useful to put standard functions into a common code repository. Traditionally, in statically defined system this simply consisted of creating object code libraries and linking to them. However, this is clearly inefficient since it means that the functionality of the system is primarily defined at compile time. Like the STANDARD APPLICATIONS(3) pattern, this pattern benefits from languages with late binding.

The main idea behind this pattern is that certain facilities need to be present for the entire time that the user is running the system. Moreover, these are facilities should be running prior to other applications being launched. This determination can be driven by a number of different forces. Since these applications are often used to provide fundamental services (remember that we want to avoid changing the backplane directly), they need to be always present. They may also have a complex or lengthy start up procedure, again making it attractive to effectively automate their start up.

We define a special class of STANDARD APPLICATIONS(3) which are loaded by the BACKPLANE(2) as part of system initialization. These applications have all the attributes associated with a STANDARD APPLICATION(3), but typically have a life span equal to that of the backplane itself. Frequently, these applications will register standard functionality via the INTERFACE REPOSITORY(4) which can subsequently accessed by any application, including another bootstrap application. There should be no restrictions on the type of functionality provided by these applications, i.e., they can be GUI-oriented, network-oriented, etc.

This construct supports developing the system on an incremental basis because it allows new high level functions to be added to the system as they are needed. Additionally, the last bootstrap application can also be the top level interface for the application. Obviously, this opens up opportunities for providing different versions of the top level interface or even varying it on a user by user basis. For example, this might be a top level menu which displays the commands that a particular user or class of users is allowed to run. Alternatively, as new features are added, rather than having to release a new version of the system, each user could simply be supplied with a new list of bootstrap applications.

Therefore:

Add new functionality to the system using applications which are automatically loaded by the backplane. Make their functionality available via the interface repository. Only modify the backplane if the functionality is related to lifecycle management.

APPENDIX A

This appendix consists of a number of Java classes which implement the concepts discussed in this pattern language. In particular, there are classes that correspond the following constructs:

- BACKPLANE
- CONFIGURATION SERVICES
- STANDARD APPLICATION
- INTERFACE REPOSITORY
- DESKTOP

In particular, the desktop construct is implemented using JavaSoft's Swing API, which supports a MDI (multiple document interface) style of windows.

BACKPLANE

```
package sample.code;

// JDK imports
import java.util.*;
import java.awt.*;
import java.awt.event.*;

// Swing imports
import com.sun.java.swing.*;

import sample.code.configuration.*;

/**
 * This class is the heart of the system and is the first component
 * to be bootstrapped. Once it has finished initializing itself,
 * it will use the <code>ConfigurationService</code> to determine
 * what <code>Bootstrap Applications</code> to load.
 * @author Nicholas Jacobs - &copy; Copyright 1998
 * Permission is granted to copy for the PLoP-98 conference
 * @version $Id: Backplane.java,v 1.1 1998/06/29 03:53:47 nick Exp $
 */
public class Backplane
    implements ApplicationListModel, InterfaceRepository, Desktop {

    private static Backplane instance=null;
    private boolean initialized=false;
    private AppManager appManager;
    private ConfigurationService configService;
    private Vector appList=new Vector();
    private SwingDesktop desktop;

    // Load our default properties via the ConfigurationService.
    private void loadDefaultProperties() throws ConfigurationException {
        String s = configService.getProperty("applications");
        if (s == null || s.length() == 0) {
            System.err.println("Warning: No applications defined.");
            return;
        }
        StringTokenizer tok = new StringTokenizer(s, ",");

        while (tok.hasMoreTokens())
            appList.addElement(tok.nextToken());

    }
    /**
     * The backplane is a <b>singleton</b>, so we provide a way for
     * all other components to get a handle to it.
     */
    public static Backplane getInstance() {
        if (instance == null)
            instance = new Backplane();
        return instance;
```

```
}

/**
 * Backplane lifecycle method -- This method should only be
 * called once (we keep track and make it a no-op if it is
 * called more than once). We instantiate all the important
 * components here such as the <code>ConfigurationService</code>
 * and the interface repository.<br>
 * Finally, we will load and initialize and <code>Bootstrap
 * Applications</code>.
 */
public synchronized void init()
    throws BackplaneException, ConfigurationException {
    if (initialized) return;

    configService = ConfigurationService.getInstance(this);
    loadDefaultProperties();

    desktop = new SwingDesktop("Sample Code");
    desktop.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            Backplane.this.stop();
            Backplane.this.destroy();
        }
    });

    appManager = new AppManager(this);
    appManager.init();
}

/**
 * Backplane lifecycle method -- This will start all the
 * applications. This method can be called more than once.
 */
public synchronized void start() { appManager.start(); }

/**
 * Backplane lifecycle method -- This will stop all the
 * applications. This method can be called more than once.
 */
public synchronized void stop() { appManager.stop(); }

/**
 * Backplane lifecycle method -- This will destroy all the
 * applications and then exit the Java VM.
 */
public synchronized void destroy() {
    appManager.destroy();
    desktop.removeAll();
    System.exit(0);
}

/**
 * Provide a standard way to launch an application.
 */
public void startApp(String name) throws BackplaneException {
    appManager.startApp(name);
}

// -- Method from ApplicationListModel --
public Enumeration getApplications() { return appList.elements(); }

// -- Method from InterfaceRepository --
public void registerInterface(Class classRef, Object impl) {
    registerInterface(classRef.getName(), impl);
}

// -- Method from InterfaceRepository --
public void registerInterface(String name, Object impl) {
    appManager.registerInterface(name, impl);
}

// -- Method from InterfaceRepository --
public void unregisterInterface(String name) {
    appManager.unregisterInterface(name);
}
```

```
    // -- Method from InterfaceRepository --
    public void unregisterInterface(Class classRef) {
        appManager.unregisterInterface(classRef.getName());
    }

    // -- Method from InterfaceRepository --
    public Object fetchInterface(String name)
        throws UnknownInterfaceException {
        return appManager.fetchInterface(name);
    }

    // -- Method from InterfaceRepository --
    public Object fetchInterface(Class classRef)
        throws UnknownInterfaceException {
        return appManager.fetchInterface(classRef.getName());
    }


    // -- Method from Desktop --
    public JComponent add(StandardApp app, JComponent c, Object constraints) {
        return desktop.add(app, c, constraints);
    }


    // -- Method from Desktop --
    public void remove(StandardApp app, JComponent c) {
        desktop.remove(app, c);
    }

    // -- Method from Desktop --
    public void removeAll(StandardApp app) { desktop.removeAll(app); }

    // -- Method from Desktop --
    public void removeAll() { desktop.removeAll(); }

    // -- Method from Desktop --
    public void setVisible(boolean f) { desktop.setVisible(f); }

    // -- Method from Desktop --
    public void setVisible(StandardApp app, boolean f) {
        desktop.setVisible(app, f);
    }

    public void setVisible(StandardApp app, JComponent c, boolean f) {
        desktop.setVisible(app, c, f);
    }

    public void showStatus(String s) { desktop.showStatus(s);}


    /**
     * The system can be run by executing this method.
     */
    public static void main(String[] args) {
        try {
            Backplane bp = Backplane.getInstance();
            bp.init();
            bp.start();
        }
        catch (Exception e) {
            System.err.println("Failed start system: "+e);
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

CONFIGURATION SERVICE

```
package sample.code.configuration;

// JDK imports
import java.net.*;
import java.io.*;
import java.util.*;
```

```
/**
 * This class defines the configuration services for the system. It
 * is a singleton and is intended to only be accessed via its
 * <code>getInstance()</code> method, which is indexed by the requesting
 * class's <code>java.lang.Class</code> object.<br>
 * When instantiated for the first time, this class expects to find
 * a file called <b>backplane.cfg</b> in the sample (logical) directory
 * as where the class file for this class resides.
 * @author Nicholas Jacobs - &copy; Copyright 1998
 * Permission is granted to copy for the PLoP-98 conference
 * @version $Id: ConfigurationService.java,v 1.3 1998/06/29 04:20:33 nick Exp $
 */
public class ConfigurationService {

    private static Properties properties;
    private Class classRef;

    ConfigurationService(Class classRef) throws ConfigurationException {
        if (properties == null) {
            InputStream in = getClass().getResourceAsStream("backplane.cfg");
            if (in == null)
                throw new ConfigurationException("Failed to find `backplane.cfg'");
            BufferedInputStream bin = new BufferedInputStream(in);

            properties = new Properties();
            try { properties.load(bin); }
            catch (IOException e) {
                throw new ConfigurationException(e.getMessage());
            }
            finally {
                try { bin.close(); }
                catch (IOException e) {}
            }
        }
        this.classRef = classRef;
    }


    /** Create an object based on the caller's <code>this</code>
     * (typically) value.
     */
    public static ConfigurationService getInstance(Object object)
        throws ConfigurationException {
        return getInstance(object.getClass());
    }


    /** Create an object based on the caller's <code>Class</code>
     * (typically) value. This is "standard" way to get a reference
     * a <code>ConfigurationService</code> object.
     */
    public static ConfigurationService getInstance(Class classRef)
        throws ConfigurationException {

        return new ConfigurationService(classRef);
    }

    /**
     * Having obtained an instance of this class, this method can
     * be used to obtain the value in question.
     */
    public String getProperty(String name) {
        return properties.getProperty(classRef.getName()+"."+name);
    }
}
```

STANDARD APPLICATION

```
package sample.code;

import com.sun.java.swing.*;

/**
 * This class defines what the methods that an application must supply
 * in order to be run by the backplane.
 *
 * Nicholas Jacobs - &copy; Copyright 1998
 * Permission is granted to copy for the PLoP-98 conference
 * @version $Id: StandardApp.java,v 1.1 1998/06/29 03:53:49 nick Exp $
 * @see Backplane
 */
public class StandardApp {

    /**
     * All applications have a handy utility function for accessing
     * the <code>Backplane</code> object.
     */
    private Backplane backplane;
    private Desktop desktop;

    /**
     * If this method is overriden, it is imperative that the
     * <code>super</code>-class should be called.
     */
    protected void init(Backplane backplane) {
        this.backplane = backplane;
        this.desktop = backplane;
    }

    /**
     * All applications have a handy utility function for accessing
     * the <code>Backplane</code> object.
     */
    public Backplane getBackplane() { return backplane; }

    public String getName() { return getClass().getName(); }

    /**
     * Lifecycle method -- Will be called only once when application
     * is loaded for the first time.
     */
    public void init() {}

    /**
     * Lifecycle method -- Will be called immediately after the
     * client's <code>init()</code> method is called. Will also
     * be called if the backplane's <code>start()</code> method
     * is invoked.
     */
    public void start() {}

    /**
     * Lifecycle method -- Will be called when the backplane's
     * <code>stop()</code> method is called. This may signal that
     * all applications are to suspend their operation.
     */
    public void stop() {}

    /**
     * Lifecycle method -- Will be called when the backplane's
     * <code>destroy()</code> method is called. This will be
     * called just prior to the backplane's shutting down.
     */
    public void destroy() {}

    /**
     * This is the top-level interface to asking an application
     * to creating a view, i.e., provide a GUI interface to it.
     * Not all applications are required to provide this capability.
     * @param arg Application-specific argument. This may be some
     * kind of memento, or indicating some kind of state message.
```

```
     */
    public void createView(Object arg) {}

    public JComponent add(JComponent c, Object constraints) {
        return desktop.add(this, c, constraints);
    }

    /** Remove a previously added component. */
    public void remove(JComponent c) {
        desktop.remove(this, c);
    }


    /** For a given application, remove all the GUI components. */
    public void removeAll(StandardApp app) {
        desktop.removeAll(this);
    }

    /** Hide or show all components for a particular application. */
    public void setVisible(boolean f) {
        desktop.setVisible(this, f);
    }

    /** Control the appearance of a particular component for
     * a particular application.
     */
    public void setVisible(StandardApp app, JComponent c, boolean f) {
        desktop.setVisible(this, c, f);
    }

    /**
     * Show a status message (if supported by implementation).
     */
    public void showStatus(String message) {
        desktop.showStatus(message);
    }

}
```

INTERFACE REPOSITORY

```
package sample.code;

/**
 * This interface defines the operations that can be done on
 * the interface repository.
 *
 * @author Nicholas Jacobs  - &copy; Copyright 1998
 * Permission is granted to copy for the PLoP-98 conference
 * @version $Id: InterfaceRepository.java,v 1.1 1998/06/29 03:53:48 nick Exp $
 */
public interface InterfaceRepository {
    /**
     * Add an interface.
     * @param name should be a unique name (typically the class name
     * of the interface itself).
     * @param impl reference to an object which implements the
     * specified interface.
     */
    void registerInterface(String name, Object impl);

    /**
     * Add an interface, but pass in the a <code>Class</code>
     * reference instead.
     * @param classRef reference to the class of the interface
     * @param impl reference to an object which implements the
     * specified interface.
     */
    void registerInterface(Class classRef, Object impl);

    /**
     * Unregister an existing interface.
     */
    void unregisterInterface(String name);

    /**
```

```
     * Unregister an existing interface.
     */
    void unregisterInterface(Class classRef);

    /**
     * Return the implementation for an interface.
     */
    Object fetchInterface(String name) throws UnknownInterfaceException;

    /**
     * Return the implementation for an interface.
     */
    Object fetchInterface(Class classRef) throws UnknownInterfaceException;

}
```

APPMANAGER

This class, while not discussed directly in this paper, is an important helper class for the backplane and provides an implementation of many important facilities, such as the interface repository and dynamic application loading.

```
package sample.code;

import java.util.*;

/**
 * This package-private class provides the implementation of the
 * important application loading and interface repository-related
 * routines. It is used heavily by the <code>Backplane</code>
 * class.
 *
 * @author Nicholas Jacobs - &copy; Copyright 1998
 * Permission is granted to copy for the PLoP-98 conference
 * @version $Id: AppManager.java,v 1.1 1998/06/29 03:53:47 nick Exp $
 */
class AppManager {
    private ApplicationListModel appListModel;
    private Hashtable appCache=new Hashtable();
    private Hashtable interfaceRepository=new Hashtable();

    AppManager(ApplicationListModel appListModel) {
        this.appListModel = appListModel;
    }

    synchronized void startApp(String appName) throws BackplaneException {
        init(appName);
        start(appName);
    }

    void init() {
        Enumeration apps = appListModel.getApplications();
        while (apps.hasMoreElements()) {
            String appName = (String)apps.nextElement();
            try { init(appName); }
            catch (Exception e) {
                System.err.println("Warning: Exception in init'ing "+appName);
                e.printStackTrace();
            }
        }
    }

    void start() {
        Enumeration apps = appListModel.getApplications();
        while (apps.hasMoreElements()) {
            String appName = (String)apps.nextElement();
            try { start(appName); }
            catch (Exception e) {
                System.err.println("Warning: Exception in start'ing "+appName);
                e.printStackTrace();
            }
        }
    }

    void stop() {
        Enumeration apps = appListModel.getApplications();
```

```
    while (apps.hasMoreElements()) {
        String appName = (String)apps.nextElement();
        try { stop(appName); }
        catch (Exception e) {
            System.err.println("Warning: Exception in stop'ing "+appName);
            e.printStackTrace();
        }
    }
}

void destroy() {
    Enumeration apps = appListModel.getApplications();
    while (apps.hasMoreElements()) {
        String appName = (String)apps.nextElement();
        try { destroy(appName); }
        catch (Exception e) {
            System.err.println("Warning: Exception in destroy'ing "+
                               appName);
            e.printStackTrace();
        }
    }
}

synchronized void init(String appName) throws BackplaneException {
    try {
        StandardApp app = (StandardApp)appCache.get(appName);
        if (app == null) {
            Class c = Class.forName(appName);
            app = (StandardApp)c.newInstance();
            appCache.put(appName, app);

            app.init(Backplane.getInstance());
            app.init();
        }
    }
    catch (ClassNotFoundException e) {
        throw new UnknownAppException("Failed to find application: "+e);
    }
    catch (InstantiationException e) {
        throw new BackplaneException("Failed to create application: "+e);
    }
    catch (IllegalAccessException e) {
        throw new BackplaneException("Failed to access application: "+e);
    }
    catch (ClassCastException e) {
        throw new BackplaneException("Does not implement StandardApp: "+e);
    }
}

synchronized void start(String appName) throws BackplaneException {
    StandardApp app = (StandardApp)appCache.get(appName);
    if (app == null)
        throw new UnknownAppException(appName);
    app.start();
}

synchronized void stop(String appName) throws BackplaneException {
    StandardApp app = (StandardApp)appCache.get(appName);
    if (app == null)
        throw new UnknownAppException(appName);
    app.stop();
}

synchronized void destroy(String appName) throws BackplaneException {
    StandardApp app = (StandardApp)appCache.get(appName);
    if (app == null)
        throw new UnknownAppException(appName);
    app.destroy();
}

void registerInterface(String ifName, Object impl) {
    interfaceRepository.put(ifName, impl);
}

Object fetchInterface(String ifName) throws UnknownInterfaceException {
    Object impl = interfaceRepository.get(ifName);
    if (impl == null)
```

```
            throw new UnknownInterfaceException(ifName);
        return impl;
    }

    void unregisterInterface(String ifName) {
        interfaceRepository.remove(ifName);
    }
}
```

SWINGDESKTOP

Finally, this is an implementation of the abstract desktop interface.

```
package sample.code;

// JDK imports
import java.awt.*;
import java.awt.event.*;
import java.util.*;

// Swing imports
import com.sun.java.swing.*;

// Framework imports
import sample.code.configuration.*;

/**
 * This is package-private helper class for the backplane. In
 * particular, it implements the desktop's functionality as
 * Swing multi-document interface.
 *
 * Nicholas Jacobs - &copy; Copyright 1998
 * Permission is granted to copy for the PLoP-98 conference
 * @version $Id: SwingDesktop.java,v 1.1 1998/06/29 03:53:49 nick Exp $
 * @see Backplane
 * @see StandardApp
 */

class SwingDesktop {
    JLayeredPane lc;
    JTextField statusMessage;
    JFrame frame;
    Hashtable components = new Hashtable();

    SwingDesktop(String name) {
        frame = new JFrame(name);
        Container cp = frame.getContentPane();
        cp.setLayout(new BorderLayout());
        lc = new JDesktopPane();
        lc.setOpaque(false);
        cp.add(lc, BorderLayout.CENTER);

        statusMessage = new JTextField("Ready.");
        statusMessage.setEditable(false);
        cp.add(statusMessage, BorderLayout.SOUTH);

        Dimension d;
        try {
            ConfigurationService cs = ConfigurationService.getInstance(this);
            String ws = cs.getProperty("width");
            String hs = cs.getProperty("height");
            Container cp = f.getContentPane();

            if (ws == null || ws.length() == 0 ||
                hs == null || hs.length() == 0)
                d = new Dimension(640, 480);
            else {
                d = new Dimension(Integer.parseInt(ws),
                                  Integer.parseInt(hs));
            }
        }
        catch (NumberFormatException e) { d = new Dimension(640, 480); }
        catch (ConfigurationException e) { d = new Dimension(640, 480); }

        frame.setSize(d);
        frame.show();
    }
```

```
void addWindowListener(WindowListener l) {
    frame.addWindowListener(l);
}


public JComponent add(StandardApp app, JComponent c, Object constraints) {
    String key = app.getName();
    Vector v = (Vector)components.get(key);
    if (v != null) {
        if (v.contains(c)) {
            c.getParent().setVisible(true);
            return (JInternalFrame)c.getParent();
        }
        else {
            JInternalFrame f = makeFrame(app, c, constraints);
            v.addElement(c);
            return f;
        }
    }
    else {
        v = new Vector();
        components.put(key, v);
        JInternalFrame f = makeFrame(app, c, constraints);
        v.addElement(c);
        return f;
    }
}

void remove(StandardApp app, JComponent c) {
    String key = app.getName();
    Vector v = (Vector)components.get(key);
    if (v != null && v.contains(c)) {
        v.removeElement(c);

        JInternalFrame f = (JInternalFrame)c.getParent();
        f.setVisible(false);
        f.dispose();

        lc.remove(f);
        lc.repaint();
    }
}

void removeAll(StandardApp app) {
    String key = app.getName();
    Vector v = (Vector)components.get(key);
    if (v != null) {
        for (int i = 0; i < v.size(); i++) {
            JComponent c = (JComponent)v.elementAt(i);
            remove(app, c);
        }
        v.removeAllElements();
        components.remove(key);
    }

}

void setVisible(boolean f) {
    Enumeration e = components.keys();
    while (e.hasMoreElements()) {
        String k = (String)e.nextElement();
        Vector v = (Vector)components.get(k);
        if (v != null) {
            for (int i = 0; i < v.size(); i++) {
                JComponent c = (JComponent)v.elementAt(i);
                c.getParent().setVisible(f);
            }
        }
    }
}

void removeAll() {
    Enumeration e = components.keys();
    while (e.hasMoreElements()) {
        String k = (String)e.nextElement();
        Vector v = (Vector)components.get(k);
```

```
            if (v != null) {
                for (int i = 0; i < v.size(); i++) {
                    JComponent c = (JComponent)v.elementAt(i);
                    c.getParent().setVisible(false);
                    if (c.getParent() instanceof JInternalFrame)
                        ((JInternalFrame)c.getParent()).dispose();
                }
                v.removeAllElements();
                components.remove(k);
            }
        }

    }

    void setVisible(StandardApp app, boolean f) {
        String key = app.getName();
        Vector v = (Vector)components.get(key);
        if (v != null) {
            for (int i = 0; i < v.size(); i++) {
                Component c = (Component)v.elementAt(i);
                c.getParent().setVisible(f);
            }
        }
    }
    void setVisible(StandardApp app, Component c, boolean f) {
        String key = app.getName();
        Vector v = (Vector)components.get(key);
        if (v != null && v.contains(c)) {
            c.getParent().setVisible(f);
        }
    }

    public void showStatus(String message) {
        statusMessage.setText(message);
    }

    // Helper function to put a JInternalFrame around an
    // application's component.
    private JInternalFrame makeFrame(StandardApp app, JComponent c,
                                     Object constraints) {

        JInternalFrame f = new JInternalFrame(app.getName());
        cp.setLayout(new BorderLayout());
        cp.add(c, BorderLayout.CENTER);
        if (constraints != null && constraints instanceof Integer)
            lc.add(f, constraints);
        else
            lc.add(f, JLayeredPane.DEFAULT_LAYER);

        f.setClosable(true);
        f.setMaximizable(true);
        f.setIconifiable(true);
        f.setResizable(true);
        f.pack();
        f.setVisible(true);
        return f;
    }
}
```