

Tweek

The Programmer's Guide

Patrick Hartling

Tweek: The Programmer's Guide

by Patrick Hartling

1.2

Published \$Date: 2007-04-18 11:28:09 -0500 (Wed, 18 Apr 2007) \$

Copyright © 2002–2007 Iowa State University

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being Appendix D, *GNU Free Documentation License*, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in Appendix D, *GNU Free Documentation License*.

Table of Contents

Preface	vii
I. Introduction	1
1. Tweek	3
C++ API Design Overview	3
Subject	3
Observer	3
CORBA Manager	4
Subject Manager	4
CORBA	4
Interface Definition Language	5
Supported Languages	5
2. JavaBeans	7
Bean Categories	7
Service Beans	7
Viewer Beans	7
Panel Beans	8
Generic Beans	8
XML	8
II. Programming	10
3. IDL	11
4. C++	13
Deriving from <code>tweek::SubjectImpl</code>	13
Using the CORBA Manager	15
Using the Subject Manager	17
Subject Manager Initialization	17
<code>tweek::SubjectManagerImpl</code> API	19
5. Java	21
GUI Library	21
Bean Library	21
Event Library	21
Network Library	21
Bean Delivery Library	22
6. Putting It All Together	23
Collaborative Slider	23
The Subject	23
The Observer	26
The Server Application	28
The JavaBean	30
Running the Example	36
File Loader	36
The JavaBean	36
XML File	40
7. C++ as a Client	41
The CORBA Service	41
Example Client Application	41
StringSubject Interface	41
StringSubject Interface Implementation	42
Observer Implementation	44
Client Application	46
8. Python	53
III. Appendices	54
A. Compiling Example Code	56
SliderSubject	56

File Loader	58
B. CORBA Implementations	59
C. Legal	60
D. GNU Free Documentation License	61
PREAMBLE	61
APPLICABILITY AND DEFINITIONS	61
VERBATIM COPYING	62
COPYING IN QUANTITY	62
MODIFICATIONS	63
COMBINING DOCUMENTS	64
COLLECTIONS OF DOCUMENTS	65
AGGREGATION WITH INDEPENDENT WORKS	65
TRANSLATION	65
TERMINATION	65
FUTURE REVISIONS OF THIS LICENSE	66
ADDENDUM: How to use this License for your documents	66
Bibliography	67
Glossary	68
Index	71

List of Examples

2.1. PlexusGraphView.xml snippet	9
2.2. Viewers.xml	9
3.1. Subject.idl	11
3.2. CustomSubject.idl	11
4.1. CustomSubjectImpl.h	13
4.2. CustomSubjectImpl.cpp	14
4.3. TweekApp.cpp	15
4.4. TweekApp.cpp	17
6.1. SliderSubject.idl	23
6.2. SliderSubjectImpl.h	24
6.3. SliderSubjectImpl.cpp	25
6.4. SliderObserverImpl.java	26
6.5. SliderSubjectApp.cpp	28
6.6. NetworkTestBean.xml	35
6.7. FileOpenTestBean.xml	40
7.1. StringSubject.idl	41
7.2. StringSubjectImpl.h	42
7.3. StringSubjectImpl.cpp	44
7.4. StringObserverImpl.h	44
7.5. StringObserverImpl.cpp	45
7.6. client.cpp: Required Header Files	46
7.7. client.cpp: Implementation of main(), Part I	47
7.8. client.cpp: Implementation of main(), Part II	48
7.9. client.cpp: Implementation of chooseSubjectManager()	50

Preface

This book is the programmer's guide for Tweak. The main focus is how to use the features and capabilities of Tweak to enable cross-platform Java graphical user interfaces (GUIs) to communicate with C++ applications.

The history of Tweak dates back to April 2000, though the basis for its existence comes from circa 1997. At Iowa State University's Virtual Reality Applications Center, the idea of using a Java GUI to communicate with, and possibly perform manipulations on, C++ applications is the foundation for dynamic re-configuration of VR Juggler. To that end, the Java application VjControl was developed specifically for that purpose. VjControl was started in 1997 and has been under development ever since.

Building on the basic idea of a Java GUI that could communicate with a C++ application, a viewer was written for the Distributed Shared Object (DSO) software system written in April 2000. This iteration of DSO was for a class project (Computer Science 552 taught by Dr. Johnny Wong). At the time, we (Allen Bierbaum and I) felt that CORBA could be used as a way to share arbitrary object-based information between applications on a network. The Java GUI was used to visualize the network of CORBA-connected nodes and to manipulate the network by disconnecting and reconnecting the nodes. In the end, CORBA was not the right solution to this problem, but the basis for network visualization was founded. Using CORBA for communication between the C++ code and Java code was relatively easy, however.

In April 2001, we were again faced with a class project. Based on the results of the CORBA-based DSO, we felt that it would be best to implement a peer-to-peer software multicasting system that would be more efficient than CORBA. This new system, called Plexus, would still offer the same capabilities of cross-platform data distribution, but we had to give up the inherent cross-language support offered by CORBA. Again, we wanted a Java GUI for visualizing the network, and we started with the DSO viewer code. Instead of using CORBA, which we had had some difficulties with the previous year, we chose to use Java's built-in Remote Method Invocation (RMI) system. While RMI is very easy to use between two Java applications, Java to C++ communication is difficult. It requires the use of the Java Native Interface (JNI) so that natively compiled C++ code can communicate in memory with a loaded Java virtual machine (JVM). All of the Java code and RMI was collected into a package called PlxView ("plex-view").

Despite the difficulty of writing JNI code, the RMI solution was effective, for the most part. In September 2001, yet another project loomed. By this time, we were fed up with RMI, and we decided to go back to CORBA solely for communicating between Java and C++. Indeed, by using CORBA, we could write network visualization software in *any* language and communicate through the same channels.

Beyond just communicating with the C++ applications, we had high hopes for using RMI to download Java code at run time to add custom visualization panels to the Java GUI dynamically. The Java code would come in the form of JavaBeans. While PlxView was designed to be modular from the start, we had not implemented the code for downloading the JavaBeans. To begin that work, I started writing code on a flight from Dallas/Ft. Worth to London, England, the evening of October 1, 2001. While at a CAVE workshop in Stockholm, Sweden, I took advantage of down time and late nights to extend PlxView to load and use JavaBeans.

When I returned to the United States after the week in Stockholm, PlxView had evolved into what could be called version 0.0.0 of the Tweak Java GUI (sans CORBA, however). Its primary purpose was still Plexus network visualization. Design discussions with other members of the Juggler Team evolved the code into what is now the Tweak Java API. The Plexus-specific parts were separated into what is now the PlxView Bean, and the remaining code was moved into the new Tweak source tree.

The only remaining piece was CORBA support. Another student, Andrew Schwantes, had been experimenting with CORBA in a smaller system, and his C++ CORBA code was used as a starting point for the Tweak C++ API. After much discussion with Allen Bierbaum, the code was re-written entirely to make use of the Observer design pattern, and the Subject Manager was added. After all of that, the Java

CORBA code was relatively trivial to add. And thus, the foundation for Tweek was in place by November 2001. By this time, it was already in use by the Fall 2001 Plexus class project.

Part I. Introduction

Table of Contents

1. Tweek	3
C++ API Design Overview	3
Subject	3
Observer	3
CORBA Manager	4
Subject Manager	4
CORBA	4
Interface Definition Language	5
Supported Languages	5
2. JavaBeans	7
Bean Categories	7
Service Beans	7
Viewer Beans	7
Panel Beans	8
Generic Beans	8
XML	8

Chapter 1. Tweek

Programmers using Tweek must first understand what it is and what it does, at least at a conceptual level. Tweek has two parts: a Java *API* and a C++ API. The two provide very distinct functionality, but they share a common bond through the use of remote method calls on shared objects. Objects defined in the C++ programming language can be accessed by objects defined in the Java programming language. Similarly, objects defined in Java can be accessed by C++ code. The C++ objects may be visualized and manipulated using the Tweek Java *graphical user interface* (GUI), a tool written using the Tweek Java API. This cross-language functionality is achieved through the use of remote method calls.

Some readers may be familiar with remote procedure calls (RPC), a remote programming system first introduced by Sun Microsystems that uses the procedural programming paradigm. Remote method calls differ primarily through the use of the object-oriented programming paradigm. Tweek is implemented in two object-oriented languages and thus lends itself very well to a system implementing remote method calls.

In the remainder of this chapter, we present a high-level description of the Tweek C++ API. The Java API design is much more complex and is not included in this chapter. (Refer to Chapter 5, *Java* for details on using the Java API.) Most users of Tweek need to know more about the C++ side of Tweek than the Java side. This is because Tweek is designed around the philosophy of a simple Java GUI interacting with a potentially complex C++ application.

C++ API Design Overview

The heart of the Tweek software system implements the Observer pattern [Gam95]. This design pattern is used to define the relationship between the Java GUI (observer) and the C++ application (subject). Within this section, we explain how the subject and observer are used. Moving beyond the subject/observer pattern, we also explain the *Subject Manager* and the *CORBA Manager*. These four components make up the entirety of the C++ design.

Subject

The subjects in Tweek are part of the C++ applications. The communication “channels” are defined by the subjects’ interfaces. An observer is attached to a subject, and whenever the state of a subject changes, it notifies all of its attached observers.

The Tweek C++ API defines the basic subject interface (`tweek::Subject`) that implements the subject pattern [Gam95]. Users of the Tweek C++ API derive from the base subject implementation (`tweek::SubjectImpl`) and extend it by adding their own interface methods. This extension is two-fold. First, an interface must be defined using the *Interface Definition Language* (IDL). Then, the interface must be implemented in C++ code. (Refer to the section called “Interface Definition Language” for more information about IDL in Tweek.)

Observer

The observers in Tweek are (traditionally) part of the Java GUI¹. They observe the state of the remote subjects and can provide a visual rendering of that state.

Programmers *do not* define interfaces for the observers. Instead, the Tweek C++ API defines a basic observer interface called `tweek::Observer`. There is no “standard” observer implementation that corresponds to `tweek::SubjectImpl`. By design, observers must correspond directly with subjects, but

¹Beginning with Tweek 0.13, helper classes for writing C++ observers are included with the C++ API. Users of older versions can make use of observers written in C++ (or any other programming language with a CORBA implementation), but the application developers will have to write the CORBA client code entirely from scratch.

there is no need to extend the basic observer interface using IDL. Observer implementations simply inherit from the basic observer class (`tweek.ObserverPOA` in Java or `POA_tweek::Observer` in C++) and implement the `update()` method. Other extensions can be added in the custom observer class, of course, but an implementation of `update()` is always required.

CORBA Manager

CORBA tends to have a high learning curve. It is a very powerful system, but that power leads to a lot of complexity. To reduce the complexity of starting and using an *ORB*, Tweek provides a CORBA Manager. Its primary function is to initialize a local ORB. It does this by creating the *Portable Object Adapter* (POA), resolving the initial reference to the Naming Service, and starting a thread for the ORB to handle requests.

Note

An explanation of the POA is beyond the scope of this book. Users of Tweek do not have to use the POA directly because the CORBA Manager and Subject Manager hide these details. Interested readers are referred to [Hen99] for more information about the POA and CORBA in general.

Once the local ORB is initialized, the Subject Manager (discussed next) must be created. This is also done through the CORBA Manager because the Subject Manager is a CORBA object. The newly created Subject Manager will be a *servant* object to which CORBA references can be created.

Refer to the section called “CORBA” for more information about CORBA and its use in Tweek. For the most part, the use of CORBA is an implementation detail. Users of the Tweek C++ API must initialize the CORBA Manager, however, and it is important to understand its place in the overall system.

Subject Manager

The Tweek Subject Manager exists to simplify the use of CORBA further. At a very high level, it acts as a simplified, specialized CORBA Naming Service. Users of Tweek register subject servants with the Subject Manager. The Subject Manager handles the CORBA registration and activation of the servants. After being registered, subjects are accessed using symbolic strings. The strings are user-defined and do not necessarily conform to any CORBA-related standard. They are, in essence, identifiers used to look up the subject within the Subject Manager's collection of known subjects.

CORBA

CORBA, the *Common Object Request Broker Architecture*, is a powerful tool for *distributed programming*. It is a *language-independent* standard specified by the Object Management Group [<http://www.omg.org/>] (OMG). Many CORBA implementations, both free and commercial, exist for a wide variety of languages (e.g., C, C++, Java, Perl, Python, and Smalltalk). CORBA allows communication between software written in any programming language running on any operating system on any hardware architecture. It handles all serialization and de-serialization of objects and method parameters so that programmers do not have to worry about endian issues and other system incompatibilities.

Before going further with the discussion of CORBA in Tweek, readers must be familiar with some terminology. In CORBA, the physical object to which references are made is called a *servant*. The servant is an instance of some class that implements an interface and derives from `CORBA::Object` (or `org.omg.CORBA.Object` in Java). The actual details of servant implementations are postponed for later sections. For now, it is important to remember that there will be an object located in physical memory on some machine, and *references* will be made to that object through CORBA. Users acquire references by looking up the object by name in what is known as the *CORBA Naming Service*. The Naming Service has objects registered within its database, and clients request references from the database. When the reference is made available, methods may be invoked on it. Since the physical object

resides in another memory space, this will create network traffic, though it is entirely transparent to the programmer.

CORBA uses *Object Request Brokers* (ORBs) to manage locally registered objects and to communicate with remote objects. The remote objects are managed by ORBs that reside locally on the machines that have the servants. Two ORBs communicate with each other using a standard protocol. In this case, that protocol is the *Internet Inter-ORB Protocol* (IIOP). IIOP is a new addition to Version 2.3 of the CORBA standard. It allows two ORBs written by different vendors to communicate and inter-operate. This capability is crucial to the correct functionality of Tweek and many other CORBA-based software systems.

Within the scope of Tweek, CORBA is used to enable transparent communication between C++ applications and the Tweek Java GUI. C++ objects registered with a local ORB are made available to the Java GUI through the Subject Manager. Beyond this, CORBA exists mostly "behind the scenes" so that developers of Tweek-based software do not have to learn very much about CORBA. Programmers must understand the Interface Definition Language, however, and this is explained next.

Interface Definition Language

The *Interface Definition Language* (IDL) is used by the CORBA standard to define the *interfaces* for remotely accessible objects. An IDL file looks very much like a simple C++ class declaration in a header file, though data members are not allowed in the interface. Thus, IDL is used exclusively to define the methods of the objects and external data structures that may be passed as arguments to those methods.

The interfaces alone are not sufficient to implement objects that may be handled by CORBA. A language-specific implementation must be written so that servants can be instantiated and registered with an ORB. To implement an interface, an *IDL compiler* must first be used to generate skeleton code for a specific language from the IDL file. Using the generated code, an implementation is then written. In Chapter 3, *IDL*, we explain in more detail how to use IDL to define interfaces.

Supported Languages

As discussed above, a very powerful feature of CORBA is its language independence. As of this writing, Tweek itself includes support for C++ and Java as the primary languages. Support for generating the stub code needed to access Tweek Subjects through Python was added in early August 2003, and a PyQt-based GUI is being written so that developers can use Python and Qt for making GUI panels instead of Java. There is no restriction, other than time and resources, that prevents the addition of support for other languages. In this section, we explain how C++, Java, and Python are used in Tweek.

C++

A key part of the overall Tweek design is that complex, high-performance applications will be written in C++. While this may not necessarily be the case in every situation, this is the assumption made for the design and implementation of the Tweek C++ API. As mentioned previously, support could be added for other languages so that they too may fulfill the role of C++ in Tweek.

C++ Subjects

Using the C++ API on the server side to create subjects, programmers activate a local ORB using the CORBA Manager. Once an ORB is available, servants that will act as subjects can be registered with the Subject Manager. The subjects are activated within the local POA by the Subject Manager, thus alleviating some work for programmers. Once activated, the subjects may be accessed remotely through CORBA by code written in any language—including C++.

C++ Observers

Using the C++ API on the client side to create observers, programmers again activate a local ORB using the CORBA Service. Once an ORB is available, servants that will act as observers can be registered

with the local POA via the CORBA Service. Once activated, the observers can be attached to remote subjects that may be written in any language. C++ observer code is very similar to Java observer code.

Java

In the Java programming language, the Swing API provides developers with a very nice suite of classes for writing cross-platform GUIs. When developing Tweek, we took advantage of Swing and JavaBeans technology [Jbe02] to write a generalized GUI framework. Users can plug components (Beans) into this framework at runtime to extend its functionality. The Beans can get access to remote C++ objects through the CORBA services provided by the Tweek Java API. Similar to the C++ API, the use of CORBA in Java has been simplified so that programmers can use it with little effort and without a comprehensive understanding of CORBA in general.

Programmers will use Java as part of their Tweek programming to write JavaBeans. Compared to the potential complexity of the GUI code for Beans, little CORBA programming must be done in Java. Beans may be as simple or as complex as necessary to meet the needs of individual projects. More information about JavaBeans is provided in Chapter 2, *JavaBeans*.

Python

With Python, we use PyQt [Pyq03] as the GUI interface. PyQt is highly portable, and a GPL version makes it easy to develop non-commercial, high-performance user interfaces. Because PyQt wraps Qt which in turn utilizes the native windowing system, user interfaces developed with PyQt tend to perform much better than Swing-based Java GUIs. Of course, natively compiled C++ that uses Qt directly would perform better still, but Python provides a degree of portability not offered (directly) by C++. We have used PyQt successfully to develop GUIs that run on desktops as well as on PDAs that include Qtopia.

Chapter 2. JavaBeans

The Tweek Java GUI uses JavaBeans to be more flexible and accessible to programmers. The GUI is a framework into which graphical and non-graphical components may be “plugged”. Graphical components add interaction functionality. Non-graphical components extend internal functionality, oftentimes needed by the graphical components. Conceptually, this follows the traditional use of plug-in architectures wherein the components are discovered dynamically and added into the larger framework. In the case of the Tweek Java GUI, the plug-ins will fit into one of four categories, the most important of which is *Panel Bean*.

Bean Categories

There are four types of Beans that may be loaded by the Tweek Java GUI. They are categorized based on functionality and what is known about them in advance. The following lists the four categories in order of decreasing *a priori* knowledge.

1. Service Beans
2. Viewer Beans
3. Panel Beans
4. Generic Beans

Service Beans

Services encapsulate functionality that may be useful to other parts of the Tweek system or dynamically loaded code. The entire interface for Service Beans must be known when the code using the service is compiled. This is because the using code needs to be able to take advantage of the service. Because Service Beans may be loaded dynamically, using code must be prepared for the case when the Bean containing the service was not found. In other words, code that uses services cannot necessarily assume that the service will be available.

Not all services are loaded dynamically as Beans. Some services are loaded statically because they are needed by core components. These include the Environment Service and the Global Preferences Service. There is a guarantee that the code for these services will always be available. This guarantee is especially important because the Tweek core needs to add information to the Environment Service at startup. The Global Preferences Service is needed to configure the overall behavior of the Tweek GUI.

Viewer Beans

Viewer Beans provide a rendering of the tree of Panel Beans (discussed next). They provide the viewer component of the model/view pattern [Gam95]. All Viewer Beans are loaded dynamically, and the active viewer can be changed at runtime by editing the global preferences. This feature is realized through the flexibility of the model/view pattern.

Viewer Beans must implement the `org.vrjuggler.tweek.beans.BeanModelViewer` interface. To simplify implementation, they may be derived from `org.vrjuggler.tweek.beans.DefaultBeanModelViewer`, a class that implements aspects of the interface that are unlikely to vary between viewer implementations. The use of the interface is needed so that the GUI frame can assume certain behaviors about the viewer.

Panel Beans

Most programmers using Tweak will write Panel Beans. These provide custom interfaces for whatever users need. In most cases, a Panel Bean will provide a graphical interface that can manipulate and/or control a C++ application, but developers are not strictly limited to this use.

Only one assumption is made about Panel Beans: the primary class for the Bean must be a subclass of `javax.swing.JComponent`. Optionally, the primary class may implement one or more publicly provided interfaces that provide the Java GUI with more information about the capabilities of the Bean. When loaded, the GUI checks to see what, if any, interfaces are implemented by the Bean. Based on the results, special actions may be taken to provide the Bean with extended functionality.

For example, Beans that can load files should implement `org.vrjuggler.tweak.beans.FileLoader`. When the Bean is focused in the viewer, the File menu will be modified to enable the Open, Save, and Close items. If the user selects one of these items, the Bean is informed and can take appropriate customized actions. The result in this case is context-specific loading and unloading of files.

Generic Beans

Nothing at all is assumed about Generic Beans. This Bean category is provided so that other Beans can do their own dynamic code loading. For example, a Bean that uses a factory pattern may want to have the “workers” loaded dynamically based on some criteria. Thus, the functionality of the factory can be changed dynamically.

The Tweak Java GUI does not use Generic Beans itself. These are provided more for users of the Tweak Java API. It is up to those programmers to decide how to handle the Generic Beans on a case-by-case basis.

XML

All JavaBeans loaded by the Tweak Java GUI are describe by at least one *XML* file. The XML file can contain information about many Beans or about a single Bean. The XML file itself is a “beanlist” document. The four Bean categories, described above, each have an XML element that has children giving information about the specific Bean. The elements are `<service>`, `<viewer>`, `<guipanel>`, and `<generic>`.

All Bean XML entries must contain a `<file>` element. Through its `source` attribute, this element provides the path to the JAR file that contains the full Bean code. When specifying the JAR file path, environment variables may be used. They must use the syntax `#{ENV_VAR}` (the curly braces are required). The `class` element gives the fully qualified name of the class stored within the JAR file that will be instantiated. The extension `.class` must not be specified. This is to allow the use of serialized classes which have the extension `.ser`. The Tweak Bean-loading code will figure out what is available and take the right actions.

In addition to the `<file>` element, a `<dependencies>` element may be specified. Dependencies of the Bean may be named as external JAR files or other Beans may be listed therein. The `<dependencies>` element may contain zero or more elements of type `<jar>` and/or `<bean>`. The `<jar>` element has a single attribute, `path`, which gives the path (a semi-colon separated list of directories) where the JAR file may be found. The contents of the `<jar>` element defines the name of the JAR file. The `<bean>` element's contents defines the name of the Bean that the current Bean depends on.

Of the four Bean categories, the XML for Panel Beans can contain the most information. In addition to the previously mentioned elements, Panel Bean entries may have two optional elements: `<tree>` and `<icon>`. The element `<tree>` specifies the path within the Bean tree hierarchy where the Panel Bean will be placed. The path is given as a `/`-separated list of directories. If the named path does not exist when the Bean is loaded, it will be created. The element `<icon>` names a custom icon for the Bean and

a tool-tip. An example of a Panel Bean XML entry is shown in Example 2.1, “PlexusGraphView.xml snippet”. Note that this is not the full file—it is only the <guipanel> element for a single Bean.

Example 2.1. PlexusGraphView.xml snippet

```
<guipanel name="Graph View">
  <file name="{PLX_BASE_DIR}/share/plexus/beans/PlexusGraphView.jar"
        class="plx.graphview.GraphView" />
  <tree path="/" />
  <dependencies>
    <jar path="{PLX_BASE_DIR}/share/plexus/beans">openjgraph.jar</jar>
    <jar path="{PLX_BASE_DIR}/share/plexus/beans">jgraph.jar</jar>
    <jar path="{PLX_BASE_DIR}/share/plexus/beans">PlexusComm.jar</jar>
  </dependencies>
  <icon source="jar:file:{PLX_BASE_DIR}/share/plexus/beans/PlexusGraphView.jar!/p
        tooltip="Plexus Network Graph Visualization" />
</guipanel>
```

This shows the use of all the elements that may be children of <guipanel>. Note that the source attribute of <icon> gets its icon image using a JAR URL.

Another example XML file is shown in Example 2.2, “Viewers.xml”. This is the actual file used to load the two Viewer Beans that come with the Tweek distribution. This is a complete file containing two Viewer Bean entries.

Example 2.2. Viewers.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beanlist xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="http://www.vrjuggler.org/tweek/xsd/1.1/be
  <viewer name="Tree Viewer">
    <file name="{TWEEK_BASE_DIR}/share/tweek/beans/Viewers.jar"
          class="org.vrjuggler.tweek.treeviewer.BeanTreeViewer" />
  </viewer>
  <viewer name="Icon Viewer">
    <file name="{TWEEK_BASE_DIR}/share/tweek/beans/Viewers.jar"
          class="org.vrjuggler.tweek.iconviewer.BeanIconViewer" />
  </viewer>
</beanlist>
```

Part II. Programming

In the following chapters, we present the basic information needed to start programming with Tweek. There will be discussion covering IDL, C++, Java, and CORBA. To use Tweek effectively, a good understanding of C++ and Java is required. The IDL aspect of Tweek is minimal, and programmers familiar with basic object-oriented concepts should be able to understand IDL code easily. Discussion related to CORBA is based on the brief introduction given in the section called “CORBA”. The Tweek Java and C++ APIs are designed to hide most CORBA details. Whenever possible, references are given to good sources of information on all of the aforementioned topics.

Before proceeding, it is important to know that Tweek is designed and implemented to work with VR Juggler 1.1 and 2.0. It can be used with VR Juggler 1.0, but there have been reports of conflicts occurring between VR Juggler 1.0 and the VR Juggler Portable Runtime (VPR) that is part of VR Juggler 1.1. The Tweek C++ code uses VPR for threading, but it is possible to replace the VPR objects with VR Juggler 1.0 thread objects.

Chapter 3. IDL

In this chapter, we present the basic information needed to define interfaces that will be used by Tweek. This is not a detailed introduction to IDL programming. Readers are referred to [Hen99].

IDL “programming” means defining *interfaces*. In the scope of Tweek and CORBA, the interfaces declare what operations may be performed on CORBA references. The arguments and return values may be of several basic types including, but not limited to, string, int, long, and float. These types are specified in a language-independent manner. When the IDL compiler generates the code for a specific language, the language-specific types that correspond to the IDL types are used.

In Example 3.1, “Subject.idl”, we show the Tweek `Subject` interface. Note the similarity to a C++ header file. The IDL file can be included by other IDL files, and thus it must “protect” the contents in the same manner as a header file. The actual `Subject` interface is defined within the `tweek` module. An IDL module corresponds to a C++ namespace or to a Java package. The interface itself has three methods: `attach()`, `detach()`, and `notify()`. The first two take a read-only argument of type `Observer`. The fact that the argument is read-only (to the server) is specified by the `in` modifier. Other modifiers are `out` (sent from server to client) and `inout` (initialized by the client, writable by the server). The third method, `notify()`, takes no arguments, and none of the methods have a return type.

Example 3.1. Subject.idl

```
#ifndef _TWEAK_SUBJECT_IDL_
#define _TWEAK_SUBJECT_IDL_

#include <tweek/idl/Observer.idl>

module tweek
{
    interface Subject
    {
        void attach(in Observer o);
        void detach(in Observer o);
        void notify();
    };
};

#endif
```

By definition, all objects are passed by reference in CORBA. The modifier stating the readability and/or writability in the IDL file determines how the referenced object may be modified, if at all, within the method.

Applications that make use of Tweek will define custom interfaces that extend the `Subject` interface. For example, consider a custom subject that maintains a floating-point value. It could have the following interface:

Example 3.2. CustomSubject.idl

```
#ifndef _CUSTOM_SUBJECT_IDL_
#define _CUSTOM_SUBJECT_IDL_
```

```
#include <tweek/idl/Subject.idl>

module mymod
{

interface CustomSubject : tweek::Subject
{
    float getValue();
    void setValue(in float v);
};

};

#endif
```

In this interface, we define two methods: `getValue()` and `setValue()`. The implementation of this interface would of course include these methods and would derive from the implementation of the `Subject` interface.

Chapter 4. C++

Writing C++ code that makes use of Tweak is not difficult, though it often requires some good planning. With the current code base, the C++ side of things maintains the state information through an implementation of the Tweak `Subject` interface. Instances of such an implementation may need to communicate with other parts of a given application, and it is important to define these relationships well. In other words, as a maintainer of application state information, the subject implementation should have easy access to that state information.

Furthermore, developers must keep in mind that there may be asynchronous execution of application code as a result of using Tweak. The local ORB runs in its own thread, and as such, it executes methods of *servants* from that thread. Whatever the servant does, it should be thread-safe with respect to the rest of the application.

In this chapter, we cover each aspect of writing C++ code that uses the Tweak API. We begin by explaining how to make a custom subject implementation. Then, we discuss the use of the CORBA Manager from user-level code. We conclude the chapter with an overview of using the Subject Manager.

Deriving from `tweek::SubjectImpl`

To create a custom subject implementation, you must derive from two classes: the abstract class that defines the custom interface and `tweek::SubjectImpl`. Referring back to the interface shown in Example 3.2, “CustomSubject.idl”, the basic C++ class declaration would appear as follows:

Example 4.1. CustomSubjectImpl.h

```
1 #ifndef _CUSTOM_SUBJECT_IMPL_H
   #define _CUSTOM_SUBJECT_IMPL_H

   #include <tweek/CORBA/SubjectImpl.h>
5 #include <CustomSubject.h>

   namespace mymod
   {

10 class CustomSubjectImpl : public POA_mymod::CustomSubject,
                           public tweek::SubjectImpl
   public:
       CustomSubjectImpl() : mValue(0.0f)
       {
15         ;
       }

       virtual ~CustomSubjectImpl()
       {
20         ;
       }

       virtual float getValue();

25       virtual void setValue(float v);

       mymod::CustomSubject_ptr _this()
       {
30         return POA_mymod::CustomSubject::_this();
       }
```

2

3
4
5

```

private:
    float mValue;
};
35 }
#endif

```

2 Here we declare our parent classes, `POA_mymod::CustomSubject` and `tweek::SubjectImpl`. The first is code generated by the IDL compiler, and the second is included as part of the Tweak C++ API. Both are necessary for this custom interface to work correctly as a Tweak subject.

4 These two declarations correspond to the `CustomSubject` interface defined in Example 3.2, “CustomSubject.idl”. The implementations of these methods are not shown here, but they are required for the code to compile. That is, the declarations in `POA_mymod::CustomSubject` are pure virtual methods, and an instance of `mymod::CustomSubjectImpl` cannot be created unless these methods are implemented.

5 Overriding the method named `_this()` is required due to the diamond inheritance tree created by deriving from `POA_mymod::CustomSubject` and `tweek::SubjectImpl`. Both of these classes derive from `tweek::Subject`. The `_this()` method plays a critical role in the CORBA communication, and it is imperative that it return the correct type to the caller when invoked on a servant instance. Without this override, the returned type will be `tweek::Subject_ptr`, and attempts to narrow to `mymod::CustomSubject_ptr` will fail.

Note the namespaces used on this method. The return type is `mymod::CustomSubject_ptr`, which corresponds to the namespace in which the classes `CustomSubjectImpl` and `CustomSubject` are defined. To get the actual value to return, `POA_mymod::CustomSubject` (one of the two parent classes) is used.

The implementations of `getValue()` and `setValue()` are fairly obvious, though they are presented here for the sake of completeness. Note, however, that `setValue()` changes the state of the subject, and thus any observers must be notified of the change. The implementations are shown in the following example.

Example 4.2. CustomSubjectImpl.cpp

```

#include <CustomSubjectImpl.h>

namespace mymod
{

float CustomSubjectImpl::getValue()
{
    return mValue;
}

void CustomSubjectImpl::setValue(float v)
{
    mValue = v;
    tweek::SubjectImpl::notify();
}

}

```

The key point to note is the call to `tweek::SubjectImpl::notify()` in the `setValue()` im-

plementation. In general, anything that modifies the state of the subject requires that this method be invoked. Note also that the method is fully qualified so that we are sure to call the correct implementation.

Using the CORBA Manager

Initializing the CORBA Manager is straightforward, but it does require exception handling. If the exceptions are not handled correctly, applications will abort if an exception is thrown but not caught. Refer to a C++ reference for more information about exceptions and exception handling in C++.

The following example shows a `main()` function for an application that performs all the Tweak initialization steps. We separate the discussion into two parts: one part for the CORBA Manager and one part for the Subject Manager (discussed in the next section).

Example 4.3. TweekApp.cpp

```

1 #include <vpr/vpr.h>
  #include <vpr/Thread/Thread.h>
  #include <vpr/Util/Debug.h>
  #include <tweek/CORBA/CorbaManager.h>
5
  #include <CustomSubjectImpl.h>

  /**
10  * This application starts the CORBA server for the C++ side
  * of the test.
  */
  int main(int argc, char* argv[])
  {
15      tweek::CorbaManager mgr;

      // The first thing we have to do is initialize the Tweek
      // CORBA Manager. If this fails, we're out of luck.
      try
20      {
          if ( mgr.init("example", argc, argv) )
          {
              bool status(false);

              // Once the CORBA Manager is initialized, we need
              // to create a Subject Manager. This will hold our
              // CustomSubject object.
              try
25              {
                  status = mgr.createSubjectManager();

30                  // If we were able to create the Subject Manager,
                  // now we register our objects with it.
                  if ( status )
                  {
35                      // First, create real instances of the C++
                      // object that will be the CORBA servant. This
                      // must be allocated on the heap.
                      mymod::CustomsubjectImpl* custom_subj =
                      new mymod::CustomSubjectImpl();

40                      // Now we try to register the subject and give
                      // it a symbolic, easy-to-remember name.
                      try
                      {
45                          mgr.getSubjectManager()->

```

1
2

3

4
5

```

        registerSubject(slider_subj, "CustomSubject");
    }
    catch (...)
    {
50         vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
            << "Failed to register subject\n"
            << vprDEBUG_FLUSH;
    }

55     // We are done with our pointer to the servant.
    slider_subj->_remove_ref();
    }
}
catch (CORBA::Exception& ex)
60 {
    vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
        << "Caught an unknown CORBA exception when "
        << "trying to register!\n" << vprDEBUG_FLUSH;
    }
65
    if ( ! status )
    {
        vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
70         << "Failed to register Subject Manager instance\n"
        << vprDEBUG_FLUSH;
    }

    std::cout << "Press 'x' to exit" << std::endl;
    char input;
75

    // Loop forever so that we can act sort of like
    // a server.
    while ( 1 )
    {
80         std::cin >> input;
        if ( input == 'x' )
        {
            break;
        }
85         else
        {
            vpr::System::msleep(100);
        }
    }
90 }
else
{
    vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
95     << "CORBA failed to initialize\n"
    << vprDEBUG_FLUSH;
}
}
catch (...)
100 {
    vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
        << "Caught an unknown exception!\n"
        << vprDEBUG_FLUSH;
    }
105 vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
    << "Exiting\n" << vprDEBUG_FLUSH;

    return 0;
}

```

- 2 These two headers are typically needed. The first includes the declaration of the Tweek CORBA Manager, and the second is the subject implementation declaration, shown in Example 4.1, "CustomSubjectImpl.h".
- 3 In order to use CORBA through Tweek, the CORBA Manager must be created and initialized. Any number of these may be created, but in general, only one is needed per application. Here, we declare an instance of `tweek::CorbaManager` on the stack.
- 5 Next, we must initialize the CORBA Manager using the method `tweek::CorbaManager::init()`. The first argument provides a unique (ideally) identifier for the local Portable Object Adapter (POA). The second and third arguments are `argc` and `argv` respectively. They come in through the argument list of `main()` and represent the command-line arguments. Any arguments relevant to ORB initialization are removed from `argv`, and `argc` is decremented accordingly (it is passed by reference).
- 6 To ensure that no exceptions go uncaught, we enclose the bulk of `main()` in a try/catch block that catches any exception. This is handled by the argument list passed to the catch block, `(...)`. This is the equivalent of catching `java.lang.Exception` in Java.

Using the Subject Manager

To demonstrate use of the Subject Manager, we begin by revisiting the `main()` function examined in the previous section. This time, we will focus our attention on the code related to the Subject Manager only. We will also explain how to use the extended API of the Subject Manager implementation.

It is important to know that the Subject Manager is a CORBA object that can be accessed by remote code. In the following example, the methods used are defined in the class `tweek::SubjectManagerImpl`, the C++ implementation of the `tweek::SubjectManager` interface. The details of how the Subject Manager is handled through CORBA are largely irrelevant for most users of Tweek. Simply bear in mind that the Subject Manager is accessibly remotely and that it simplifies the use of CORBA in general.

Subject Manager Initialization

In order to use the Tweek Subject Manager, it must be initialized. Each CORBA Manager should have a single Subject Manager associated with it. If not, use of Tweek will be much more difficult because the CORBA Manager and the Subject Manager together hide most of the details relating to the use of CORBA. The following example shows how to initialize the Subject Manager using the CORBA Manager object created earlier.

Example 4.4. TweekApp.cpp

```

1 #include <vpr/vpr.h>
  #include <vpr/Thread/Thread.h>
  #include <vpr/Util/Debug.h>
  #include <tweek/CORBA/CorbaManager.h>
5
  #include <CustomSubjectImpl.h>
  /**
   * This application starts the CORBA server for the C++ side of
10 * the test.
   */
  int main(int argc, char* argv[])
  {
15     tweek::CorbaManager mgr;

        // The first thing we have to do is initialize the Tweek
        // CORBA Manager. If this fails, we're out of luck.

```

```

try
{
20   if ( mgr.init("corba_test", argc, argv) )
      {
          bool status(false);

          // Once the CORBA Manager is initialized, we need to
          // create a Subject Manager. This will hold our
          // CustomSubject object.
          try
          {
30             status = mgr.createSubjectManager();

                // If we were able to create the Subject Manager,
                // now we register our objects with it.
                if ( status )
                {
35                     // First, create real instances of the C++
                        // object that will be the CORBA servant. This
                        // must be allocated on the heap.
                        mymod::CustomsubjectImpl* custom_subj =
                        new mymod::CustomSubjectImpl();
40
                            // Now we try to register the subject and give
                            // it a symbolic, easy-to-remember name.
                            try
                            {
45                                 mgr.getSubjectManager()->
                                    registerSubject(custom_subj,
                                                "CustomSubject");
                            }
                            catch (...)
                            {
50                                 vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
                                    << "Failed to register subject\n"
                                    << vprDEBUG_FLUSH;
                            }

55                                 // We are done with our pointer to the servant.
                                    custom_subj->_remove_ref();
                            }
                        }
                    catch (CORBA::Exception& ex)
                    {
60                         vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
                            << "Caught an unknown CORBA exception when "
                            << "trying to register!\n"
                            << vprDEBUG_FLUSH;
                    }

                    if ( ! status )
                    {
70                         vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
                            << "Failed to register Subject Manager instance\n"
                            << vprDEBUG_FLUSH;
                    }

75                     std::cout << "Press 'x' to exit" << std::endl;
                        char input;

                        // Loop forever so that we can act sort of like
                        // a server.
80                     while ( 1 )
                        {

```

1

2

3

4

```

        std::cin >> input;
        if ( input == 'x' )
        {
85             break;
        }
        else
        {
90             vpr::System::msleep(100);
        }
    }
}
else
{
95     vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
        << "CORBA failed to initialize\n" << vprDEBUG_FLUSH;
}
}
catch (...)
100 {
    vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
        << "Caught an unknown exception!\n" << vprDEBUG_FLUSH;
}
105 vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
    << "Exiting\n" << vprDEBUG_FLUSH;
return 0;
}

```

- 1 After the COBRA Manager has been initialized successfully, the Tweek Subject Manager must be created. We do this by invoking the method `tweek::CorbaManager::createSubjectManager()` on our `tweek::CorbaManager` instance.
- 2 Once we have a valid Subject Manager, we must register subjects with it in order for object references to be passed out by CORBA. This creates the *servant* to which `mymod::CustomSubject` references will be made.
- 3 Once the servant is created, it is registered with the Subject Manager. The Subject Manager will activate the servant within the POA so that references to it can be created and returned to clients. We register it with the symbolic name “CustomSubject” that can be referenced later by remote objects.
- 4 After the subject is registered, all the work is done. This simple application now just waits for clients to request references. It will exit when the user enters 'x'.

Note that all the code relating to the Subject Manager is enclosed within another try/catch block. This block only catches exceptions of type `CORBA::Exception`. Anything more general is caught by the larger try/catch block.

tweek::SubjectManagerImpl API

The class `tweek::SubjectManagerImpl` has some methods that are not part of the IDL-specified interface. One such method is `registerSubject()`, which was used in the preceding example. Some other methods that may be of interest to users are described in the following subsections.

setApplicationName()

```
public void setApplicationName(const std::string& appName);
```

This method can be used to set one of the application-specific identifiers within the Subject Manager.

Namely, it sets the application name identifier. These identifiers are used to aid users in choosing a Subject Manager instance when making remote connections to applications.

setUserName()

```
public void setUserName(const std::string& userName);
```

Similar to `setApplicationName()`, this method allows users to tell the Subject Manager their user name. When multiple users are all running the same application (and are thus using the same parameter to `setApplicationName()`), this provides another level of uniqueness. If this method is not used, the Subject Manager will try to get the user name through the `$USER` environment variable.

addInfoItem()

```
public void addInfoItem(const std::string& key,  
                        const std::string& value);
```

If the previous two built-in Subject Manager identifiers are not enough, the method `addInfoItem()` allows users to define their own unique identifiers. The first parameter is the identifier, and the second is its (ideally) unique value. Users are free to add any key/value pairs they need in order to aid in the selection of a Subject Manager at runtime.

Chapter 5. Java

In this chapter, we present the libraries that make up the Tweek Java API. In general, we only provide a general overview of the libraries. Interested readers are referred to the *Tweek Java Programmer's Reference* for more comprehensive documentation.

The Tweek Java API is broken up into a collection of Java class libraries, each packaged in a unique JAR file. Programmers can choose the library or libraries they need when writing Java code that uses Tweek. It is possible to write JavaBeans that do not use any of the Tweek Java libraries, but such Beans may not be very full-featured. On the other hand, because the Tweek Java GUI is capable of loading any JavaBean, there is no reason that a fully functional Bean must be written to take advantage of Tweek services and utilities.

GUI Library

The GUI Library contains the heart of the Tweek Java GUI. Indeed, this library is what contains the Tweek Java GUI, and it makes use of all the other Tweek Java libraries to do its job. In general, there is not much code in this library that is of interest to Bean developers. The lone exception is the singleton `org.vrjuggler.tweek.gui.MessagePanel`. This class provides access to the message panel at the bottom of the Tweek Java GUI layout. Tweek Bean authors can make use of this singleton to post messages to that panel for users of the Tweek GUI to see. In most cases, this should be the preferred method of printing status messages rather than using `System.out.print*()` or `System.err.print*()` simply because users of the GUI may not have easy access to the console to see the messages printed there. The GUI Library is found in `$TWEЕК_BASE_DIR/share/tweek/java/Tweek.jar`.

Bean Library

The Bean Library is provided to help simplify the process of loading and managing JavaBeans. The internal handling of JavaBeans performed by the Tweek Java GUI uses classes in this library to communicate with other objects when Beans are loaded, instantiated, removed, etc. The Bean Library is found in `$TWEЕК_BASE_DIR/share/tweek/java/TweekBeans.jar`.

Event Library

The Event Library is used by the GUI Library (i.e., by the Tweek Java GUI) to inform Panel Beans about GUI events. This includes events such as GUI iconification, GUI closing, and the gaining and losing of focus by the GUI. The Event Library is found in `$TWEЕК_BASE_DIR/share/tweek/java/TweekEvents.jar`.

An interesting part of the Event Library is the Event Listener Registry. Historically, the Tweek Java GUI has performed automatic, run-time registration of Beans as event listeners based on the interface(s) that the Beans implement. That is not always convenient, however. For example, a Panel Bean author may want to use utility classes as event listeners rather than the Panel Bean itself. In this case, the utility classes can be registered manually with the Tweek Java GUI using the Event Listener Registry.

Network Library

The Network Library abstracts the use of CORBA by the Tweek Java GUI¹. It includes the CORBA Service class that handles all ORB and remote reference management, and it has all the Java classes gen-

¹This single library is perhaps the reason that people feel compelled to use the “pre-fab” Tweek Java GUI rather than write their own GUI in the language and widget set of their choice.

erated by the IDL-to-Java compiler when the Tweek Java API was compiled. Beyond this, it includes the event listener interface for Tweek Beans that wish to be notified of network connection and disconnection events. The Network Library is found in `$TWEЕК_BASE_DIR/share/tweek/java/TweekNet.jar`.

Bean Delivery Library

The Bean Delivery Library makes use of the Bean Library and the Network Library to allow JavaBeans to be transferred across the network. More specifically, a remote subject, most likely written in C++, can “push” JavaBeans to the Tweek GUI where they can be handled as though they were loaded from the local disk. The Bean Delivery Library is found in `$TWEЕК_BASE_DIR/share/tweek/java/TweekEventDelivery.jar`.

Chapter 6. Putting It All Together

Based on the information presented in the previous chapters, we can combine everything into an examples. In this chapter, we present the step-by-step process for using the Tweek Java and C++ APIs.

Collaborative Slider

In this example, we explain how to develop a simple Tweek interface. The goal is to have a “collaborative” slider in a Java GUI component. The value displayed by the slider is retained by a C++ application so that multiple independent sliders can show the same value. The steps explained here are highly representative of the normal steps to be followed when using the Tweek Java and C++ APIs. The structure of the following sections lays out the order of the steps taken. An example makefile that goes along with the code presented can be found in the section called “SliderSubject” of Appendix A, *Compiling Example Code*. The full source for the examples presented in this section can be found in `$TWEAK_BASE_DIR/share/test/NetworkTestBean`.

The Subject

To begin, the subject interface must be defined in IDL and implemented in C++. The interface itself will be “compiled” into Java and C++ code. Both ends of the communication channels must know the interface in order for the references to be used, thus requiring the generation of code for both languages.

Creating the Interface

Creating an IDL interface involves writing an IDL file. For this example, we will be storing an integer variable in a C++ servant. The Java GUIs will need to read and write the value, so we need two methods: `getValue()` and `setValue()`. The type being passed between *ORB*s will be long, a 32-bit integer. Depending on the target language, this will map to the corresponding type of the same size.

Example 6.1. SliderSubject.idl

```
1 #ifndef _NETWORK_TEST_SLIDER_SUBJECT_IDL_
2 #define _NETWORK_TEST_SLIDER_SUBJECT_IDL_
3
4 #include <tweek/idl/Subject.idl>
5
6 module networktest
7 {
8     interface SliderSubject : tweek::Subject
9     {
10         void setValue(in long val);
11         long getValue();
12     };
13 };
14
15 #endif
```



IDL files are run through the C preprocessor so that they may include external files. To prevent including the same source multiple times, the file must be enclosed in the traditional preprocessor protection block. This is exactly what is done in C/C++ header files.

All Tweek subject IDL files must include `tweek/idl/Subject.idl`. This is required so that the interface being defined can inherit from `tweek::Subject`.

4 Typically, using a module to contain the interface is recommended. Using a module defines a Java package and a C++ namespace (named `networktest` in this case).
 5 The definition of the subject interface must inherit from `tweek::Subject`. If this is not done,
 7 there is no way that the subject interface will plug into the Subject Manager.
 This simple interface has only two functions: `setValue()` and `getValue()`. The method `getValue()` takes a single read-only parameter of type `long`; `getValue()` returns a value of the same type.

The file `SliderSubject.idl` must be “compiled” by an *IDL compiler*. For use with Tweek, the interface must be compiled into Java and C++ code. The generated Java code will be used solely for communicating with CORBA `networktest.SliderSubject` references. The generated C++ code will be extended to provide an implementation of the `networktest::SliderSubject` interface. (The implementation will be a CORBA *servant* object to which references will be made by remote Java code.)

Implementing the Interface in C++

After running an IDL compiler to generate the stub CORBA code, the interface must be implemented. In particular, there will be pure virtual methods in `SliderSubject.h` that must be implemented. The implementing class will be the CORBA *servant* holding the actual data visualized in the Java GUI slider.

Example 6.2. SliderSubjectImpl.h

```

1 #ifndef _SLIDER_SUBJECT_IMPL_H_
2 #define _SLIDER_SUBJECT_IMPL_H_
3
4 #include <tweek/tweekConfig.h>
5
6 #include <vector>
7
8 #include <tweek/CORBA/SubjectImpl.h>
9 #include <SliderSubject.h>
10
11 namespace networktest
12 {
13     /**
14      * This class is an extension to the base Tweek SubjectImpl class.
15      * It uses multiple inheritance with that class and with the
16      * generated CORBA class corresponding to the IDL for
17      * SliderSubject.
18      */
19
20     class SliderSubjectImpl
21         : public POA_networktest::SliderSubject
22         , public tweek::SubjectImpl
23     {
24     public:
25         SliderSubjectImpl()
26             : tweek::SubjectImpl(), mValue(0)
27         {
28             /* Do nothing. */ ;
29         }
30
31         virtual ~SliderSubjectImpl()
32         {
33             /* Do nothing. */ ;
34         }
35

```

```

    /**
     * Sets this subject's internal value.
     */
40 virtual void setValue(long value);
    /**
     * Returns this subject's internal value.
     */
45 virtual long getValue();
    /**
     * This overriding method is needed so that the correct type
     * is returned when the _this() method is invoked. Without
     * this method, an object of type tweek::Subject_ptr would
50 * be returned.
     *
     * XXX: It may be possible to remove this requirement in
     * the future.
     */
55 networktest::SliderSubject_ptr _this()
    {
        return POA_networktest::SliderSubject::_this();
    }

60 private:
    long mValue;    /**< Our value */
    };
    } // End of networktest namespace
65

#endif /* _SLIDER_SUBJECT_IMPL_H_ */

```

5

6

7

8

1 These files will always be included by implementations of Tweek subject derived classes. The first contains the declaration for the basic Tweek subject implementation. The second contains the C++ code generated from the Tweek Observer .idl file.

2 This header is generated by the IDL compiler from SliderSubject .idl. In particular, it defines the class from which networktest::SliderSubjectImpl must inherit.

3 SliderSubject .idl, shown in Example 6.1, “SliderSubject.idl”, the interface is in the networktest module. In the C++ implementation code, the module name corresponds to a namespace.

4 The interface implementation class must inherit from the IDL-generated class POA_networktest::SliderSubject and from tweek::SubjectImpl.

6 POA_networktest::SliderSubject defines two pure virtual methods that must be implemented. These correspond to the methods in SliderSubject .idl.

7 At the time of this writing, all subject implementations must contain an overriding version of the _this() method. This is due to the use of multiple inheritance. Note the return type and the return statement. These will vary for each subject implementation based on the name of the IDL-defined interface.

8 This is the actual value being stored by the C++ servant.

In SliderSubjectImpl .h, the most important parts to note are the use of multiple inheritance, the declarations of the SliderSubject interface methods, and the implementation of _this(). The implementations of setValue() and getValue() are shown next in Example 6.3, “SliderSubjectImpl.cpp”.

Example 6.3. SliderSubjectImpl.cpp

```

1 #include <vpr/Util/Debug.h>
  #include <SliderSubjectImpl.h>
                                     1
   namespace networktest
5 {
   void SliderSubjectImpl::setValue(long value)
   {
     mValue = value;
10     // Notify any observers that our value has changed. This is very
     // important.
     tweek::SubjectImpl::notify();
15 }
   long SliderSubjectImpl::getValue()
   {
     return mValue;
20 }
   } // End networktest namespace
                                     4

```

- 1 Include the class declaration file, shown in Example 6.2, “SliderSubjectImpl.h”.
- 2 When invoked, the remote caller will pass a long value, and this saves the result into the servant's storage.
- 3 Because the subject's state has been modified, all attached observers must be notified. This is a very important step that must be taken in this method. Note that it invokes the `notify()` method of the parent class.
- 4 Observers will invoke this method when requesting the current `mValue`.

The Observer

The observer does not define its own specialized IDL interface. Instead, it makes use of the existing Tweak basic observer (`tweek.ObserverPOA` in Java). The method `update()` must be implemented. The remainder of the observer implementation is centered around communication with a `SliderSubject` object reference. All observer code is written in Java. The only C++ code for observers is part of the Tweak library, and it is generated entirely by the *IDL compiler*.

Implementing the Observer in Java

For every subject interface defined in IDL, a corresponding observer class must be written in Java. Without an observer, there is no way for the Java and C++ sides to conduct useful two-way communication. At best, the Java GUI could request a subject reference and manipulate the C++ application through the reference, but the communication would be entirely one-way.

In Example 6.4, “SliderObserverImpl.java”, we show the complete Java implementation of an observer corresponding to the `tweek::SliderSubject` interface defined earlier. (The JavaBean that uses this observer is explained in the section called “The JavaBean”.) The main focus of this observer is to update its contained `JSlider` whenever the state of the corresponding subject changes.

Example 6.4. SliderObserverImpl.java

```

1 package networktest;

   import javax.swing.DefaultBoundedRangeModel;
   import javax.swing.JSlider;

```

```

5 import tweek.*;

/**
 * Implementation of the Observer side of the Tweak
 * Subject/Observer pattern. It must extend tweek.ObserverPOA
10 * so that instances of this class can be registered as CORBA
 * servants. In addition, CORBA references to the servants must
 * be capable of being attached to remote subjects.
 */
public class SliderObserverImpl extends ObserverPOA
15 {
    public SliderObserverImpl(JSlider slider,
                               SliderSubject subject)
    {
        mSlider      = slider;
20     mSliderSubject = subject;
    }

    /**
     * Implements the required method in tweek.ObserverPOA. The
25     * remote subject will invoke this method whenever it is
     * notified of a change.
     */
    public void update()
    {
30         // If we have a valid slider object, we need to update
        // its value to whatever our subject has.
        if ( mSlider != null )
        {
            DefaultBoundedRangeModel model =
35             (DefaultBoundedRangeModel) mSlider.getModel();
            model.setValue(mSliderSubject.getValue());
            mSlider.repaint();
        }
40     }

    /**
     * Detaches this observer from our subject. This is needed
     * when shutting down a CORBA connection.
     */
45     public void detach()
    {
        mSliderSubject.detach(this._this());
    }

50     private SliderSubject mSliderSubject = null;
    private JSlider        mSlider        = null;
}

```

1
2**3****4****5**

1 As an observer, this class must derive from `tweek.ObserverPOA`. This class is generated by the IDL compiler and is part of the Tweak Network Library (see the section called “Network Library”).

2 The constructor for this observer takes two arguments: a `JSlider` object reference and a `net.tweektest.SliderSubject` object reference. The observer needs the latter argument so that it can query state information from the subject when notified of state changes. This is part of the subject/observer design pattern [Gam95].

3 As stated, all observers must implement `update()`. This will be invoked by the remote subject when its `notify()` method is invoked.

4 To get the updated state of the remote subject, the encapsulated subject reference's `getValue()` method is invoked. The value returned will be the most up-to-date information from the subject.

5 It may be convenient for the observer to implement a `detach()` method (though the name may vary). This is used when the Java application is shutting down to ensure that the remote subject

does not have dangling references to observers that no longer exist. The only action required here is invoking the subject's `detach()` method to inform the subject that this observer is going away.

This example demonstrates that observers do not have to be complex to be usable. While this example is purposely simple, it should illustrate that developers of observers do not necessarily have to make their implementations complicated. As will be shown in the section called “The JavaBean”, the JavaBean that uses this observer completes the picture and provides users with a GUI slider that can be manipulated by any number of simultaneous users.

The Server Application

Now that we have the subject and observer ready to go, we can make an application that uses them. The following example is a (relatively) simple C++ application that starts the CORBA Manager, creates the Subject Manager, registers a `networktest::SliderSubject` servant, and then waits for the user to press 'x' to exit. The use of exceptions may appear unfamiliar to some C++ programmers. CORBA makes use of exceptions as a cross-language mechanism to report errors, and thus, there must be proper exception handling code for the application to work correctly.

Example 6.5. SliderSubjectApp.cpp

```

1  #include <tweek/CORBA/CorbaManager.h>
   #include <vpr/Thread/Thread.h>
   #include <vpr/Util/Debug.h>
2
5  #include <SliderSubjectImpl.h>
   /**
   * This application starts the CORBA server for the C++ side
   * of the test.
10 */
   int main(int argc, char* argv[])
   {
   15     tweek::CorbaManager mgr;
       // The first thing we have to do is initialize the Tweek
       // CORBA Manager.  If this fails, we're out of luck.
       try
       {
20         if ( mgr.init("corba_test", argc, argv) )
           {
               bool status(false);
               // Once the CORBA Manager is initialized, we need to
               // create a Subject Manager.  This will hold our
25             // SliderSubject object.
               try
               {
                   status = mgr.createSubjectManager();
30
                   // If we were able to create the Subject Manager,
                   // now we register our objects with it.
                   if ( status )
                   {
35                       // First, create real instances of the C++
                       // object that will be the CORBA servant.  This
                       // must be allocated on the heap.
                       networktest::SliderSubjectImpl* slider_subj =
                           new networktest::SliderSubjectImpl();

```

1

2

3

4

5

6

```

40         // Now we try to register the subject and give
         // it a symbolic, easy-to-remember name.
         try
         {
45             mgr.getSubjectManager()->
                 registerSubject(slider_subj,
                               "SliderSubject");
         }
         catch (...)
         {
50             vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
                 << "Failed to register subject\n"
                 << vprDEBUG_FLUSH;
         }

55         // We are done with our pointer to the servant.
         slider_subj->_remove_ref();
     }
 }
 catch (CORBA::Exception& ex)
 {
60     vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
         << "Caught an unknown CORBA exception when "
         << "trying to register!\n"
         << vprDEBUG_FLUSH;
65 }

     if ( ! status )
     {
70         vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
                 << "Failed to register Subject Manager instance\n"
                 << vprDEBUG_FLUSH;
     }

     std::cout << "Press 'x' to exit" << std::endl;
75     char input;

     // Loop forever so that we can act sort of like
     // a server.
     while ( 1 )
80     {
         std::cin >> input;
         if ( input == 'x' )
         {
85             break;
         }
         else
         {
             vpr::System::msleep(100);
90         }
     }
 }
 else
 {
95     vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
         << "CORBA failed to initialize\n" << vprDEBUG_FLUSH;
 }
 }
 catch (...)
 {
100    vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
            << "Caught an unknown exception!\n" << vprDEBUG_FLUSH;
 }

vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)

```

7

8

```

105         << "Exiting\n" << vprDEBUG_FLUSH;
        return 0;
    }

```



These two headers are typically needed. The first includes the declaration of the Tweak CORBA Manager, and the second is the subject implementation declaration, shown in Example 6.2, "SliderSubjectImpl.h".

In order to use CORBA through Tweak, the CORBA Manager must be created and initialized. Any number of these may be created, but in general, only one is needed per application.

After the CORBA Manager has been initialized successfully, the Tweak Subject Manager must be created.

Once we have a valid Subject Manager, we must register subjects with it in order for object references to be passed out by CORBA. This creates the *servant* to which `networktest::SliderSubject` references will be made.

Once the servant is created, it is registered with the Subject Manager. The Subject Manager will activate the servant within the POA so that references to it can be created and returned to clients.

After the subject is registered, all the work is done. This application now just waits for clients to request references. It will exit when the user enters 'x'.

The application shown in Example 6.5, "SliderSubjectApp.cpp" is purposely simple. There are many ways to use the CORBA Manager and the Subject Manager. For example, an object registry could be built on top of the Subject Manager so that only specific types of servant objects may be registered. Servant registration could be automated in object constructors. The C++ API is intended to be simple to enhance its usability and flexibility, and the application shown in the example is just that: an example.

The JavaBean

We have finally reached the point at which we implement a JavaBean that can visualize the numeric data held by the C++ slider subject. Such a JavaBean must be defined as a Tweak Panel Bean. (Refer back to Chapter 2, *JavaBeans* if these statements seem unfamiliar or confusing.)

The JavaBean shown in the following example is typical of one that uses the Tweak Network library to take advantage of CORBA facilities. The code for the Bean is longer than previous examples, and because of this, it will be split into multiple code blocks. Each will be discussed in turn. The full code is in one file: `NetworkTest.java`.

Please note that the details of setting up the GUI elements used by the Bean are left out. The code in this case was generated by JBuilder and could easily vary from Bean to Bean. For the full code, please refer to the aforementioned locations.

Imports

The slider JavaBean uses common Java Swing classes, a CORBA exception class, the Tweak Event library, the Tweak Network library, and the Java code generated by an IDL compiler. The following explains how each of these are imported into the main Bean class.

```

package networktest;

import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import org.omg.CORBA.BAD_PARAM;
import org.vrjuggler.tweak.event.*;
import org.vrjuggler.tweak.net.*;
import org.vrjuggler.tweak.net.corba.*;
import tweak.*;

```



- 1 This Bean is in a package corresponding to the IDL module defined in `SliderSubject.idl`.
- 2 Using a consistent name throughout makes the code easier to manage and understand.
- 3 These imports bring in common Swing and AWT GUI classes. Different Beans will use different packages and packages.
- 4 In order to do proper narrowing of reference types, the exception `org.omg.CORBA.BAD_PARAM` must be caught.
- 5 These imports bring in the classes of the Tweak Event and Network libraries.
- 6 Tweak Java code generated by an IDL compiler will always be in the package `tweak`. This corresponds to the IDL module and to the C++ namespace.

Class Declaration

Now we can begin writing the Bean code. Besides the class `SliderObserverImpl`, this Bean has only one public class: `NetworkTest`. It will provide the GUI representation of the numeric data. The declaration of the class and its package-visible helper follow.

```
/**
 * This is an example of a JavaBean that Tweak can load dynamically. It holds
 * a JSlider that acts as an Observer in the Tweak CORBA Subject/Observer
 * pattern implementation.
 *
 * @version 1.0
 */
public class NetworkTest
    extends JPanel
    implements CommunicationListener
{
    ...
}

class FrameListener extends TweakFrameAdapter
{
    ...
}
```

- 1 As a Tweak Panel Bean, there must be a class that derives from `javax.swing.JComponent` or some subclass thereof. In this case, the superclass is `javax.swing.JPanel`.
- 2 Since this Bean needs to access the remote Subject Manager, it needs to know when a CORBA service is available. By implementing `org.vrjuggler.tweak.net.CommunicationListener`, the Bean will be informed by the Tweak Java GUI whenever the communication state with a CORBA service changes.
- 3 This Bean also listens for `TweakFrameEvents` so that it can shut itself down cleanly. To be informed of such events, a helper class that extends `org.vrjuggler.tweak.event.TweakFrameAdapter` exists.

Member Variables

Before delving into the methods of the `networktest.NetworkTest` class, it will be helpful to review the member variables used throughout the class. Refer back to this section if there is any confusion regarding the use or the type of some member variable in the method implementations.

```
private BorderLayout mBeanLayout = new BorderLayout();

private JPanel mSliderPanel = new JPanel();
private JSlider mDataSlider = new JSlider();

private SliderObserverImpl mSliderObserver = null;
```

- 1 This is the containing layout for the entire Bean.

3 The JSlider used for visually representing the subject's numeric data is `mDataSlider`, and it
 4 will be contained within `mSliderPanel`.
 The `networktest::SliderSubject` observer will be stored within `mSliderObserver`.
 1 That the observer is stored in a member variable initialized to null. It will be assigned a value when
 a CORBA service becomes available and the corresponding subject can be requested. The object itself is
 stored as a member variable so that it can be accessed by all the methods of the class.

Handling CORBA Communications

The most complex part of this Bean is the handling of CORBA communication events delivered by the Tweak GUI. All of the handling in this example will be done in the methods `connectionOpened()` and `connectionClosed()`. The steps that must be followed are straightforward, but there are errors that must be handled properly. It is the error handling that can make the code look daunting, not the use of the Tweak Network library.

```

1 /**
   * Implements the Tweak CommunicationListener interface needed
   * for being informed of new connections with remote ORBs.
   */
5 public void connectionOpened(CommunicationEvent e)
   {
   // The first thing to do is get the CORBA service object from
   // the event. We need this so we know to whom we are are
   // connecting. Once we have the CORBA service, we get its
10 // Subject Manager since that's what contains the actual
   // subjects we need.
   CorbaService corba_service = e.getCorbaService();
   SubjectManager mgr = corba_service.getSubjectManager();

15   Subject subject = mgr.getSubject("SliderSubject");
   SliderSubject slider_subject = null;

   // Try to narrow the Subject object to a SliderSubject object.
   // If this fails, it throws a CORBA BAD_PARAM exception. In
20 // that case, we open a dialog box saying that the narrowing
   // failed.
   try
   {
25     slider_subject = SliderSubjectHelper.narrow(subject);
   }
   catch (BAD_PARAM narrow_ex)
   {
30     JOptionPane.showMessageDialog(
       null,
       "Failed to narrow subject to SliderSubject",
       "SliderSubject Narrow Error",
       JOptionPane.ERROR_MESSAGE
     );
   }

35   // Ensure that slider_subject is a valid object just to be
   // safe.
   if ( slider_subject != null )
   {
40     // First, we need a Java object that implements the
   // Observer. That object must be registered with the Java
   // CORBA service.
   mSliderObserver =
     new SliderObserverImpl(mDataSlider, slider_subject);
45   corba_service.registerObject(mSliderObserver,
     "SliderObserver");

```

```

// Now that the observer is registered, we can attach it
// to the subject. The subject needs to know who its
50 // observers are so that it can notify them of updates.
   slider_subject.attach(mSliderObserver._this());

// Now we set the slider in our GUI to be whatever value
// the remote subject is holding for us.
55 mDataSlider.setValue(slider_subject.getValue());
   mDataSlider.addChangeListener(
       new SliderChangeListener(slider_subject)
   );
60 }

/**
 * Implements the Tweak CommunicationListener interface needed
 * for being informed when existing ORB connections are closed.
65 */
public void connectionClosed(ConnectEvent e)
{
    if ( mSliderObserver != null )
    {
70         mSliderObserver.detach();
        mSliderObserver = null;
    }
}

```

9

10

11

12

13

1 If the event delivered to the Bean is a newly opened connection, there is a new CORBA service
2 available, so we will create a new observer to communicate through that service.

3 The first step that must be taken is retrieving the
4 `org.omg.vrjuggler.tweak.net.corba.CorbaService` object from the event. This is
5 provided as the basis for all further actions. With the CORBA service reference, we can request the
6 Subject Manager reference. The result is a CORBA reference to the remote Subject Manager, a
7 servant within the C++ application.

8 Through the Subject Manager, we request subjects using symbolic names. In this case, we request
9 the subject with the hard-coded name "SliderSubject".

10 The call to `getSubject()` returns a reference of type `tweak.Subject`, but we need to nar-
11 row it to `networktest.SliderSubject` so we can use it. Among the code generated by the
12 compiler is a class named `networktest.SliderSubjectHelper`. Its `narrow()`
13 method is used to narrow the CORBA reference type to a more specific type. If the narrowing fails,
an exception of type `org.omg.CORBA.BAD_PARAM` is thrown, and it is best to handle it here.

Once we have the subject reference, we create an observer servant that we will attach to the sub-
ject. This is where `networktest.SliderObserverImpl` comes into the picture.

The observer servant must be registered with the local CORBA service so that references to it can
be created. This single statement takes care of all the POA activation behind the scenes.

Now that the servant is registered with the CORBA service, a reference to it can be passed to the
remote subject in order to attach the observer to the subject.

Since the initial references are passed around, we need to update the GUI slider to the current value
provided by the subject.

As part of the collaborative slider, we need to register a change listener with the GUI slider so that
it will be informed of changes made by the local user. These changes are reported to the remote
subject so that other users get the update. The class `SliderChangeListener` is shown later.

When the event delivered to the Bean was the closing of an existing connection, the CORBA service is
being shut down. When the CORBA service is shut down, we need to detach the local observer
from the remote subject to prevent further update notification attempts for this observer.

It is important to note that the use of hard-coded subject names is not recommended. In this example, the subject name is hard-coded for simplicity. The Tweak Subject Manager allows accessing code to request a list of all registered subjects. Using this information, it is possible to present the Java GUI user with a list of subjects from which they can make a selection.

Handling Frame Events

This Bean makes an effort to shut down open CORBA connections when the Tweak GUI is closed. It does this through a helper class that extends `org.vrjuggler.tweek.event.TweakFrameAdapter`, overriding only one method: `frameClosing()`. This method calls back into the `NetworkTest` class, invoking its own `frameClosing()` method. For more information about the other Tweak GUI frame events delivered through the interface `org.vrjuggler.tweek.event.TweakFrameListener`, refer to the Tweak Java API reference.

```
public class NetworkTest
    extends JPanel
    implements CommunicationListener
{
    ...
    public boolean frameClosing()
    {
        disconnect();
        return true;
    }
    ...
}

class FrameListener extends TweakFrameAdapter
{
    public FrameListener(NetworkTest bean)
    {
        this.bean = bean;
    }

    public boolean frameClosing(TweakFrameEvent e)
    {
        return bean.frameClosing();
    }

    private NetworkTest bean = null;
}
```

1 This class overrides only the method invoked when the Tweak GUI frame is closing. If this method returns false, then the GUI frame will not close.

2 The handling of the frame closing event is delegated to `NetworkTest.frameClosing()`, which in turn calls `NetworkTest.disconnect()` to shut down the CORBA connection cleanly.

For the helper class `FrameListener` to be notified of Tweak GUI frame events, it must be registered with the Tweak event listener registry (a singleton). This step is performed in the `NetworkTest` constructor, as shown below.

```
public NetworkTest()
{
    try
    {
        jbInit();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }

    mFrameListener = new FrameListener(this);
    EventListenerRegistry.instance().
```

```

        registerListener(mFrameListener,
                        TweakFrameListener.class);
    }

```

Handling Local Slider Change Events

A helper class is used to handle local change events in the slider. The class is a private inner class within the Bean and is defined as follows:

```

private class SliderChangeListener implements ChangeListener
{
    public SliderChangeListener(SliderSubject subject)
    {
        mSliderSubject = subject;
    }

    public void stateChanged(javax.swing.event.ChangeEvent e)
    {
        JSlider source = (JSlider) e.getSource();

        if ( ! source.getValueIsAdjusting() )
        {
            mSliderSubject.setValue(source.getValue());
        }
    }

    private SliderSubject mSliderSubject = null;
}

```

1
2

2 Within the `stateChanged()` method, we check to see if the user is done adjusting the slider. If the remote subject is updated so that its internal value matches that of the local slider. Other updates are automatically updated when this happens.

XML File

With the Bean implementation done, the XML file that describes the Bean must be written. All Beans loaded by the Tweak Java GUI are described by an XML file. In Example 6.6, “NetworkTestBean.xml”, we show the XML file for the Bean we have been developing.

Example 6.6. NetworkTestBean.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beanlist
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://www.vrjuggler.org/tweak/beans/1.1/beanlist" />
  <guipanel name="Network Tester">
    <file
      name="{TWEAK_BASE_DIR}/share/tweek/beans/NetworkTestBean.xml"
      class="networktest.NetworkTest" />
    <tree path="/Beans" />
  </guipanel>
</beanlist>

```

1
2
3
4
5

1 Well-formed XML requires this line.
2 Tweak XML Bean files are beanlist documents. There must be exactly one beanlist per file, but the list itself can contain multiple Beans. In this element, we also provide the information

needed to validate this document using the XML Schema `beanlist.xsd`. In this case, we use the schema that does not employ XML namespaces. Use of the schema for validation is optional, but it is recommended.

4 The Bean we have been developing is a GUI Panel Bean, and this indicates that fact. To find the Bean, a path must be specified. In this example, we will put the XML and JAR files in `$TWEAK_BASE_DIR/share/tweek/beans`. Note the use of curly braces around the environment variable. This is required. The class attribute names the class that will be instantiated, and in this case, that is `networktest.NetworkTest`. This name corresponds more to the fully qualified name of the class stored within the JAR file.

5 Panel Beans are loaded into a tree data structure and visualized using a viewer. This specifies the path within the tree using a UNIX-style path. The path is arbitrary and will be constructed as needed when the Bean is loaded.

After the Bean is compiled into a JAR file, the JAR file and XML file need to be copied into `$TWEAK_BASE_DIR/share/tweek/beans`. This is the default path that Tweek searches for Beans at startup, and it is a convenient place to put this example Bean.

Running the Example

With all the coding done and the code compiled, we can run the C++ application and connect multiple instances of the Tweek Java GUI to it. The steps to run the C++ application are as follows:

1. Run a CORBA Naming Service. This is required so that the C++ and Java ORBs can resolve symbolic references.
2. Since Tweek uses omniORB for a C++ ORB (see Appendix B, *CORBA Implementations* for more information on this), the environment variable `$OMNIORB_CONFIG` must be set. This gives the full path to an omniORB configuration file that omniORB will load at runtime.
3. Run the C++ application.
4. Run the Tweek Java GUI. This will find and load all the Beans in `$TWEAK_BASE_DIR/share/tweek/beans`.
5. Within the Java GUI, connect to the CORBA Naming Service started in step 1. The way the Bean is written, it will request the subject held by the C++ application automatically, and the slider will be updated to the current value.

File Loader

Not everything written in Tweek must use Java, C++, IDL, and CORBA. The Tweek Java GUI is a generalized Bean-loading environment, and as such, its main focus is to load JavaBeans and present them to the user. Those Beans may take advantage of CORBA, but they are not required to do so. In this section, we show another sample Bean that is much simpler than the previous example.

The Bean shown in this example can open and close multiple text files. To do this, the Bean implements the interface `org.vrjuggler.tweek.beans.FileLoader`. The Java code will be shown in separate pieces to make it easier to understand. The full code is in `$TWEAK_BASE_DIR/share/test/FileOpenTestBean/fileopentestbean/FileOpenTestBean.java`. An example makefile can be found in the section called “File Loader”. within Appendix A, *Compiling Example Code*.

The JavaBean

As in the previous example, we focus on the imported classes first. They are as follows:

```
package fileopentestbean;

import java.awt.*;
import java.io.File;
import java.io.FileInputStream;
import javax.swing.*;
import org.vrjuggler.tweek.services.ExtensionFileFilter;
import org.vrjuggler.tweek.beans.FileLoader;
```

1
2
3
4
5
6
7

1 This Bean will be in the package fileopentestbean.
 2 These are the imports of standard Java classes. In this Bean, we need two Java file I/O classes so
 3 we may load files.
 4 This line imports the Tweek Service Bean known as the ExtensionFileFilter. This
 5 provides a simplified mechanism for making file filters typically used with JFileChooser in-
 6 stances.
 7 This line imports the interface from the Tweek Bean library that we will implement.

1 We show the class declaration and the member variables.

```
public class FileOpenTestBean
    extends JPanel
    implements java.io.Serializable, FileLoader
{
    // Methods ...

    private int openFileCount = 0;

    private BorderLayout mMainLayout = new BorderLayout();
    private JLabel mBeanTitle = new JLabel();
    private JTabbedPane mTextContainer = new JTabbedPane();
}
```

1
2
3
4

1 As the primary class for a Panel Bean, this class must from javax.swing.JComponent or
 2 its subclass thereof. In this case, the class will be javax.swing.JPanel.
 3 JavaBeans are expected to implement java.io.Serializable, and this class does that. The
 4 more interesting interface for this example is
 5 org.vrjuggler.tweek.beans.FileLoader. By implementing this interface, the Tweek
 6 Java GUI will know that this Bean can manage files.

1 This is a property of the class that keeps track of the number of currently open files.
 2 This Bean is capable of opening multiple files, and it will manage them with a JTabbedPane.

1 Now that we have the basics for the class, we can start implementing the FileLoader interface. Of
 2 the methods that must be implemented, the most complex is openRequested(). It will be shown last
 3 because of its length. We will begin instead with the simplest methods.

```
1 public String getFileType()
    {
        return "Text";
    }
5 public boolean canOpenMultiple()
    {
        return true;
    }
10 public boolean canSave()
    {
        return false;
    }
```

1
2
3

```

    }
15 public boolean saveRequested()
    {
        return false;
    }
20 public boolean closeRequested()
    {
        mTextContainer.remove(
25         mTextContainer.getSelectedComponent()
        );
        openFileCount--;
        return true;
    }

30 public int getOpenFileCount ()
    {
        return openFileCount;
    }

```

4

5

6

1 This returns the type of file (or files) that can be loaded by the Bean. In this case, we are just loading plain text files.

2 This method is used to determine if the Bean can open multiple files at one time. This Bean can because of its tabbed pane.

3 This method is used to determine if the Bean can save files, and this Bean does not.

4 Because this Bean does not save files, it simply returns false if a save is requested.

5 When a close is requested, the currently selected tab is removed and the open file count is decremented.

6 This simply returns the number of currently open files.

These are all the methods of the FileLoader interface except openRequested(). We are now ready to move on to it.

```

1 public boolean openRequested()
    {
        // Initialize this to false since a lot of things can go wrong
        // in the process of opening files. Once the file is opened
5        // and read successfully, this can be changed to true.
        boolean opened = false;

        JFileChooser chooser = new JFileChooser();
        chooser.setMultiSelectionEnabled(false);
10        chooser.setDialogTitle("Text File Loader");
        chooser.setSelectionMode(JFileChooser.FILES_ONLY);

        // Only load .txt files.
        ExtensionFileFilter filter =
15        new ExtensionFileFilter("Text Files");
        filter.addExtension("txt");
        chooser.addChoosableFileFilter(filter);

        int status = chooser.showOpenDialog(this);
20        if ( status == JFileChooser.APPROVE_OPTION )
        {
            File file = chooser.getSelectedFile();

25            if ( file.canRead() )
            {
                try
                {
                    // Read the contents of the file into a byte[]

```

1

2

3

4

```

30         // object.
        FileInputStream input_file =
            new FileInputStream(file);
        byte[] file_data = new byte[(int) file.length()];
        input_file.read(file_data);
35
        // Create a text area to hold the contents of the
        // file.
        JTextArea text_area = new JTextArea();
        text_area.setEditable(false);
40        text_area.insert(new String(file_data), 0);

        // Create a scroll pane to hold the text area; add
        // it to the tabbed pane with all the other
        // previously loaded scroll panes; and make the new
        // scroll pane the selected component.
45        JScrollPane text_comp = new JScrollPane(text_area);
        mTextContainer.add(text_comp, file.getName());
        mTextContainer.setSelectedComponent(text_comp);

50        // Our work is done.
        openFileCount++;
        opened = true;
    }
    catch (java.io.FileNotFoundException ex)
55    {
        JOptionPane.showMessageDialog(
            null,
            "Cannot find '" + file.getAbsolutePath() + "'",
            "Read Error", JOptionPane.ERROR_MESSAGE
60        );
    }
    catch (java.io.IOException ex)
    {
        JOptionPane.showMessageDialog(
65            null,
            "Error reading from '" + file.getAbsolutePath() +
            "':" + ex.getMessage(),
            "Read Error", JOptionPane.ERROR_MESSAGE
70        );
    }
    else
    {
75        JOptionPane.showMessageDialog(
            null,
            "Cannot read from file '" +
            file.getAbsolutePath() + "'",
            "Read Error", JOptionPane.ERROR_MESSAGE
80        );
    }
}

return opened;
}

```

5

6

7

8

1 First, we create a `JFileChooser` that will be used to select a text file. It is configured to allow selection of only a single file each time.

2 Next, we create a filter for `.txt` files that will be used by the file chooser.

3 With those steps taken, we now open the chooser and wait for the user to select a file.

4 If a readable file was chosen, we open it and read its contents into an array of bytes.

5 In the bytes read, we can now put the contents of the text file into a read-only `JTextArea` ob-

7 Next, we put the `JTextArea` in a `JScrollPane` so that long files can be viewed more easily. The scroll pane is added to the tabbed pane and made the currently selected panel.

8 At this point, we are done, so we increment the number of open files and set the return value to `true` to indicate that a file was opened successfully.

This Bean uses no CORBA code and does not require C++ code to act as a peer. This may be the case for many Panel Beans written for Tweek. Of course, this Bean could be extended to open files that are then handed off to C++ code through CORBA.

XML File

The XML file for this Bean is very simple. It simply lists the Bean file information and puts the Bean at the root of the Bean tree. The full file is shown in Example 6.7, “FileOpenTestBean.xml”.

Example 6.7. FileOpenTestBean.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beanlist
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://www.vrjuggler.org/tweek/xsd/1.1/beanlist"
  <guipanel name="File Open Test Bean">
    <file
      name="{TWEAK_BASE_DIR}/share/tweek/beans/FileOpenTestBean.jar"
      class="fileopentestbean.FileOpenTestBean" />
    <tree path="/" />
  </guipanel>
</beanlist>
```

Chapter 7. C++ as a Client

As of Tweak version 0.13.0, users can write C++ clients in addition to Java and Python clients. As of this writing, the interface is relatively low-level. The C++ counterpart of the Java class `org.vrjuggler.tweek.net.corba.CorbaService` is used to communicate with a remote C++ server. The class is `tweek::CorbaService`, and its API is almost identical to the Java version. The basic idea is that users familiar with the Java API can easily make use of the C++ version and vice versa.

In this chapter, we explain how to write C++ clients. First, we cover the CORBA Service, and then we show how to use it with a C++ client application. The basic ideas are identical to those used in Java clients. The syntax varies slightly due to the differences in the C++ and Java mappings of IDL. Additionally, we do not need to be concerned with JavaBeans because we will be speaking strictly in terms of C++. There is nothing to prevent an ambitious user from developing a C++ counterpart to the Tweak Java GUI, however.

The CORBA Service

The CORBA Service, implemented in the class `tweek::CorbaService`, simplifies the use of CORBA for C++ clients. Instances are constructed by providing the host name and port where the CORBA Naming Service is listening for incoming requests. Once constructed, the CORBA Service must be initialized. This creates the local root POA, acquires a reference to the Naming Service, and starts the local ORB thread. At this point, the CORBA Service can be used exactly as it would be used in Java (see Chapter 6, *Putting It All Together*). For example, the remote Subject Manager reference can be requested and servants (C++ implementations of `tweek::Observer`) can be registered.

Example Client Application

To understand how to use the C++ client API, we will examine a simple application. Similar to the Java/C++ application we reviewed in Chapter 6, *Putting It All Together*, this application makes use of an IDL-specified interface, and a C++ server. The key differences, then, are that there is no Java code in this case, and we will not need to use XML.

All the code shown here, including a GNU makefile, can be found in the directory `$TWEAK_BASE_DIR/share/tweek/test/CxxClient`. In the following sections, we will refer to the specific files and highlight key sections within them. We assume that readers have already read and understood Chapter 6, *Putting It All Together*. This is important because we will refer back to concepts illustrated in that chapter. Further, we will not present the server code in this chapter because it is nearly identical to the code for the server discussed previously. Interested readers can review the file `server.cpp`.

StringSubject Interface

As usual, we begin by defining an interface. For this example, we will use an interface that provides access to a simple string object. The server (Subject) will maintain the value of the string, and the clients (Observers) will be able to query and manipulate the string. The interface will be called `CxxClientTest::StringSubject`. The code for the interface is shown in Example 7.1, “StringSubject.idl”.

Example 7.1. StringSubject.idl

```
#ifndef _CXX_CLIENT_TEST_STRING_SUBJECT_IDL_
#define _CXX_CLIENT_TEST_STRING_SUBJECT_IDL_
```

```

#include <tweek/idl/Subject.idl>

module CxxClientTest
{
    interface StringSubject : tweek::Subject
    {
        void setValue(in string val);
        string getValue();
    };
};

#endif /* _CXX_CLIENT_TEST_STRING_SUBJECT_IDL_ */

```

1
2
3

As with all user-defined Subjects, we derive our interface from `tweek::Subject`. These methods define the accessors for the encapsulated string value. Clients will invoke these to get the value and to manipulate the state of the remote Subject.

1
2
3

StringSubject Interface Implementation

Now that we have our interface defined abstractly, we must provide a C++ implementation of the Subject. This will be done in the files `StringSubjectImpl.cpp` and `StringSubjectImpl.h`. We begin with the header file, shown in Example 7.2, “StringSubjectImpl.h”. While the example code may appear long, there is only slight variation from previous Subject implementation header files. We highlight the important bits, of course.

Example 7.2. StringSubjectImpl.h

```

1  #ifndef _STRING_SUBJECT_IMPL_H
   #define _STRING_SUBJECT_IMPL_H

   #include <tweek/tweekConfig.h>
5
   #include <string>

   #include <vpr/vpr.h>
   #include <vpr/Sync/Mutex.h>
10 #include <tweek/CORBA/SubjectImpl.h>
   #include <StringSubject.h>

   namespace CxxClientTest
   {
15
   /**
    * This class is an extension to the base Tweek SubjectImpl
    * class. It uses multiple inheritance with that class and
    * with the generated CORBA class corresponding to the IDL for
20 * StringSubject.
    */
   class StringSubjectImpl
       : public POA_CxxClientTest::StringSubject
       , public tweek::SubjectImpl
25 {
   public:
       StringSubjectImpl() : tweek::SubjectImpl(), mValue("")
       {
           /* Do nothing. */ ;
30 }

```

1

2

```

    virtual ~StringSubjectImpl()
    {
        /* Do nothing. */ ;
35  }

    /**
     * Sets this subject's internal value.
     */
40  virtual void setValue(const char* value);

    /**
     * Returns this subject's internal value.
     */
45  virtual char* getValue();

    /**
     * This overriding method is needed so that the correct
     * type is returned when the _this() method is invoked.
     * Without this method, an object of type tweek::Subject_ptr
     * would be returned.
     */
50  CxxClientTest::StringSubject_ptr _this()
    {
55      return POA_CxxClientTest::StringSubject::_this();
    }

private:
    std::string mValue;          /**< Our value */
60
    /** A mutex to protect mValue accesses */
    vpr::Mutex mValueLock;
    };
65 } // End of CxxClientTest namespace

#endif /* _STRING_SUBJECT_IMPL_H_ */

```

3

4

5

6

7

- 1 We must be sure to include the header that defines the base class from which we will derive our implementation. Remember that this code will be generated by the IDL compiler.
- 2 As usual, we declare our implementation so that it derives from the class generated by the IDL compiler (POA_CxxClientTest::StringSubject) and from the basic Tweak Subject implementation class (tweek::SubjectImpl).
- 3 These are the pure virtual methods defined by POA_CxxClientTest::StringSubject that we must implement. As in previous examples, this is where we will do the work of accessing the encapsulated string value.
- 4 As with all C++ Subject implementations, we must override the method _this() in order to return the correct type. This is critical for proper use of the specific Subject type.
- 5 Here we have the string value that will be encapsulated within the servant instance. For convenience, we use the std::string type, though it will require careful handling of the char* data that CORBA actually passes around.
- 6 In this Subject implementation, we are being more paranoid about concurrent accesses to mValue, so we will protect it with a mutex.
- 7

Next, we look at the very simple implementations of CxxClientTest::StringSubjectImpl::setValue() and CxxClientTest::StringSubjectImpl::getValue(). These are found in the file StringSubjectImpl.cpp, and the code is shown in Example 7.3, “StringSubjectImpl.cpp”. It is important to note the use of guards in the method implementations. These provide an exception-safe mechanism for controlling access to the data member mValue. When constructed, the guard locks the mutex passed as the argument to the constructor. When the guard goes out of scope, the mutex is automatically unlocked.

Example 7.3. StringSubjectImpl.cpp

```

1 #include <vpr/Sync/Guard.h>
  #include <StringSubjectImpl.h>

  namespace CxxClientTest
5 {

  void StringSubjectImpl::setValue(const char* value)
  {
10     {
        vpr::Guard<vpr::Mutex> val_guard(mValueLock);
        mValue = std::string(value);

        // Notify any observers that our value has changed. This
15     // is very important.
        tweek::SubjectImpl::notify();
    }

  char* StringSubjectImpl::getValue()
20 {
        vpr::Guard<vpr::Mutex> val_guard(mValueLock);
        return CORBA::string_dup(mValue.c_str());
    }

25 } // End CxxClientTest namespace

```

1

3

4

6

As usual, we must include our header file to get the necessary declarations. Within this scoped block, we lock the `mValueLock` using a guard and assign a new value to `mValue`. Upon exiting this block, the guard goes out of scope, and `mValueLock` is unlocked. In all cases where we modify the state of the Subject, we must notify any Observers who are attached to us. Just as the comment notes, this is very important. To retrieve the current value, we return the current value of `mValue`. We use a guard again to simplify unlocking the mutex (it happens automatically upon return). Note that we make a copy of the memory contained in `mValue` using `CORBA::string_dup()`. Making a copy is required by CORBA.

1
3
4
6**Observer Implementation**

With the Subject implemented, we now turn our attention to a C++ Observer implementation. Its implementation will be very simple. Its `update()` method will query the current string value and print it to the console. The goal here is to demonstrate how to write a C++ Observer, not how to write an *interesting* Observer.

Example 7.4. StringObserverImpl.h

```

1 #ifndef _STRING_OBSERVER_IMPL_H_
  #define _STRING_OBSERVER_IMPL_H_

  #include <tweek/CORBA/Observer.h>
5 #include <StringSubject.h>

  class StringObserverImpl : public POA_tweek::Observer
  {
  public:

```

2
3

```

10   StringObserverImpl(CxxClientTest::StringSubject_var subject)
      : mSubject(subject)
      {
          /* Do nothing. */ ;
      }
15   virtual ~StringObserverImpl()
      {
          /* Do nothing. */ ;
      }
20   virtual void update();

      void detach()
      {
25     mSubject->detach(_this());
      }

private:
      CxxClientTest::StringSubject_var mSubject;
30 };

      #endif /* _STRING_OBSERVER_IMPL_H_ */

```

4

5

6

- 2 First, we include the headers we need. The first declares the basic `tweek::Observer` C++ interface. The second includes the `tweek::StringSubject` interface. We need this header so that we can hang onto a reference to our Subject.
- 3 Our `Observer` implementation derives from the basic `Tweek Observer` class `POA_tweek::Observer`. The Java equivalent is `tweek.ObserverPOA`. Refer to Example 6.4, “SliderObserverImpl.java” in the section called “Implementing the Observer in Java” to see the correspondence.
- 4 The `update()` method must be implemented for all subclasses of `POA_tweek::Observer`. Here, we just declare the method; the implementation will be shown in Example 7.5, “StringObserverImpl.cpp”.
- 5 The `detach()` method is provided as an easy way to detach the `Observer` from its `Subject`. Ideally, this would be done in the destructor so that the detaching process happens automatically when the `Observer` servant is deleted. Doing so leads to incorrect deactivation of the servant, however. Instead, we provide this method as a way to perform the detachment before servant deactivation or deletion occurs.
- 6 Finally, we have a member variable that we will use to retain a reference our `Subject`. This allows the `Observer` to be passed around to functions without passing its `Subject` explicitly. Further, we will see that having a reference to the `Subject` is required for the `update()` method implementation to work.

Now, we show the implementation of `StringObserverImpl::update()`. This method is implemented in `StringObserverImpl.cpp`. While the method body is very short, we use the `.cpp` file to encourage the implementation of methods outside of the class declaration.

Example 7.5. StringObserverImpl.cpp

```

#include <iostream>
#include <StringObserverImpl.h>

void StringObserverImpl::update()
{
    char* cur_value = mSubject->getValue();
    std::cout << "Current string value is now '"
               << cur_value << "'\n" << std::endl;
}

```

1

3

```
    delete cur_value;
}
```

4

As usual, we have to include our header file to get the class declaration information. Since `update()` is only called when the state of the Subject changes, we need to query its current state. Once we get the current string value back, we print it to the console. The value is enclosed in single quotes so that any leading or trailing whitespace is displayed clearly. Lastly, we delete the memory returned by `CxxClientTest::StringSubject::getValue()`. The burden for freeing this memory is on the client, as per standard CORBA operating procedure.

Client Application

Finally, we look at the C++ client application. This brings everything together and makes use of the `tweek::CorbaService` class to contact the remote server. The application shown below is more complicated than the server because more work must be done. We must initialize the local CORBA Service (the local ORB); we must pick out the correct Subject Manager reference; and we must get the correct `CxxClientTest::StringSubject` reference from the Subject Manager. Once all of those steps are completed, we can create an Observer servant (an instance of `StringObserverImpl`) and attach it to the remote Subject.

The file containing the complete client application source is `client.cpp`. We will examine it in three parts: the required headers, the implementation of `main()`, and the Subject Manager lookup.

Example 7.6. `client.cpp`: Required Header Files

```
#include <iostream>
#include <string>

#include <vpr/vpr.h>
#include <vpr/Util/Debug.h>
#include <tweek/Client/CorbaService.h>
#include <tweek/CORBA/SubjectManager.h>

#include <StringObserverImpl.h>
```

1
2
3

The key to the C++ client interface is the CORBA Service. Its declaration is found in `tweek/Client/CorbaService.h`. As in Java clients, the Subject Manager plays a vital role in getting references to remote Subjects. Note that the file we include here is generated by the IDL compiler and provides only the basic interface. We will not be using the actual Subject Manager C++ implementation here. That is only used on the server side. Lastly, we include the implementation of our Observer. We will be creating an instance of this class to act as the servant for an Observer reference. The servant we create will be attached to a remote Subject, just as in all Java clients.

Next, we look at the implementation of the application's `main()` function. This is where the bulk of the work is done. Of course, in real-world use, modularizing the code would be much better than dumping most of it in `main()`. For the purposes of this example, we can get by with having most of the code in `main()`.

Note the use of try/catch blocks in the client application code. As in the case of server applications (refer to the section called “The Server Application”), we are careful about handling exceptions properly. Remember that CORBA uses exceptions extensively to indicate errors, and thus it is necessary for user

code to catch them.

Example 7.7. client.cpp: Implementation of main(), Part I

```

1 int main(int argc, char* argv[])
  {
    std::string ns_host, iiop_ver;
    vpr::Uint16 ns_port;
5
    std::cout << "Naming Service host: ";
    std::cin >> ns_host;

    std::cout << "Naming Service port (usually 2809): ";
10   std::cin >> ns_port;

    std::cout << "IIOP version (usually 1.0): ";
    std::cin >> iiop_ver;

15   // Create the local CORBA Service using the Naming Service
    // URI information we just collected.
    tweek::CorbaService corba_service(ns_host, ns_port,
                                       iiop_ver);

20   try
    {
      // Attempt to initialize the CORBA Service.
      if ( corba_service.init(argc, argv) )
25         {
          // This will hold the reference to the Subject Manager
          // we use.
          tweek::SubjectManager_var subj_mgr =
            chooseSubjectManager(corba_service);

30         // Verify that we actually got a Subject Manager
          // reference back from chooseSubjectManager.
          if ( ! CORBA::is_nil(subj_mgr) )
          {
35             // Request the Subject with which we will communicate.
            // This hard-coded Subject name is not necessarily a
            // good thing.
            tweek::Subject_var subj =
              subj_mgr->getSubject("StringSubject");

40             // If the Subject Manager knows about the Subject
            // named above, then we are good to go.
            if ( ! CORBA::is_nil(subj) )
            {
45                 ... // Shown in the next example block
            }
          }
          // We did not get a Subject Manager reference back for
          // some reason.
          else
50         {
            vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
              << "No Subject Manager chosen--exiting.\n"
              << vprDEBUG_FLUSH;
          }
55     }
    // The CORBA Service initialization failed.
    else
    {
      vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)

```

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

```

60         << "CORBA Service failed to initialize\n"
           << vprDEBUG_FLUSH;
        }
    }
    catch (...)
65     {
        vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
            << "Caught an unknown exception!\n" << vprDEBUG_FLUSH;
    }
70     return 0;
    }

```

3 Here, we query input from the user to get Naming Service information. We need the host name where the Naming Service is running, the port on which it is listening, and the version of *IIOP* to use. Using the values we get from the user, we can construct an instance of `tweek::CorbaService`. Internally, it will use the values to create a URI for looking up the Naming Service reference.

4 Once we have all the necessary initialization pieces, we can construct an instance of `tweek::CorbaService`. Simply creating an instance of this class does not perform any CORBA-related activities. That is the next step.

5 Now we initialize the local CORBA Service. We pass in `argc` and `argv` in case the user provided any CORBA-specific command-line parameters. (Any such parameters will be stripped from `argv`, and `argc` will be decremented accordingly.) If initialization succeeds, we proceed to requesting the Subject Manager reference.

7 During the CORBA Service, we choose the Subject Manager through which all Subject requests will be handled. The work for making this choice is offloaded to the helper function `chooseSubjectManager()`, shown in Example 7.9, “client.cpp: Implementation of `chooseSubjectManager()`”. For now, we just assume that we got back some reference, possibly nil, to a remote `tweek::SubjectManager` object. Since we do not know for sure what the state of things is on the server side or what decision the user made, we verify that we did not get back a nil reference. If we got back a valid Subject Manager reference, we can use it just as we would in a Java client to request Subject references by name.

9 Next, we request the reference to the remote Subject. The symbolic name we use here is the same as that specified in the server application (not shown in this chapter). Note that using a hard-coded name in this way is not recommended, but we use it here for the sake of simplicity. Once we have a reference, we verify that it is not nil before trying to use it.

We now narrow our attention to the handling of the Subject reference that was returned by the Subject Manager. The code shown in Example 7.8, “client.cpp: Implementation of `main()`, Part II” comes from the “...” block in Example 7.7, “client.cpp: Implementation of `main()`, Part I”. At this point in the application execution, we know that we have a non-nil `tweek::Subject` reference, so we need to narrow it to our specific type, create an Observer servant, and attach it to the remote Subject.

Example 7.8. client.cpp: Implementation of `main()`, Part II

```

1 StringObserverImpl* string_observer(NULL);
  PortableServer::ObjectId_var observer_id;

  try
5  {
    // Attempt to narrow subj to the more specific reference type
    // CxxClientTest::StringSubject_var. If this fails, an
    // exception will be thrown and caught below.
    CxxClientTest::StringSubject_var string_subj =
10     CxxClientTest::StringSubject::_narrow(subj);

```

1

```

// Request the current value before we create the Observer.
// In this way, we can see the persistent state maintained
// by the Subject.
15 char* cur_value = string_subj->getValue();
   std::cout << "Current string value is '" << cur_value << "'
   << std::endl;
   delete cur_value;

20 // Create our Observer servant.
   string_observer = new StringObserverImpl(string_subj);

   // Register the newly created servant with our ORB's POA.
   observer_id =
25     corba_service.registerObject(string_observer,
                                   "StringObserver");

   // This could be done in the StringObserverImpl constructor,
   // but we do it here in this example just to make it clear
30 // that it is important.
   string_subj->attach(string_observer->_this());

   const std::string exit_string("Q");
   std::string cur_string;
35
   for ( ;; )
   {
       std::cout << "Enter a string (Q to quit): ";
       std::cin >> cur_string;
40
       if ( exit_string != cur_string )
       {
           string_subj->setValue(cur_string.c_str());
       }
45     else
       {
           break;
       }
   }
50 }
catch (...)
{
    vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
    << "Caught an unknown exception during object interaction!\n"
55     << vprDEBUG_FLUSH;
}

// We're done, so now we have to clean up after ourselves.
// The order of operations here is important.
60 if ( NULL != string_observer )
{
    string_observer->detach();
    corba_service.unregisterObject(observer_id);
    delete string_observer;
65 }

```

2

3

4

5

6

9

- 1 Given a non-nil Subject reference, we now need to narrow it to our specific Subject type, `StringSubject`. If the narrowing fails, an exception will be thrown. In that case, we are done because we did not get back a reference we can use.
- 2 If the narrowing succeeded, then we can create an Observer servant (an instance of our class `StringObserverImpl`). In our local ORB, this servant will handle accesses from remote objects—namely, the Subject to which it is attached.
- 3 Next, we register our newly created Observer servant with the POA in the local CORBA Service. We give it an arbitrary name, and it gives us a unique identifier for the servant. We will need this

identifier later when the application is shutting down and cleaning up after itself.

4 At long last, we can attach our Observer to the remote Subject. The mechanism for doing this is nearly identical to the Java version. The only difference is the use of the `->` operator, which is not present in Java. Once the Subject knows about our Observer, it will be informed of all state changes.

5 In this loop, we ask the user for input and then pass that input to the Subject's `setValue()` method. We do this until the user enters the string "Q", which denotes that s/he wants to quit the application.

6 This is where we modify the state of the remote Subject. We pass the C string version of the user-specified string as the argument to `CxxClientTest::StringSubject::setValue()`.

9 Once the user has requested to quit the application, we need to clean up the servant we constructed earlier. First, we detach the Observer from the remote Subject. Next, we unregister our servant using the ID returned by the local CORBA Service. Finally, we delete the heap memory allocated for the servant. With that, we are done.

The last part of the client application is the choice of the Subject Manager reference to use. In this example, we put that code in the helper function `chooseSubjectManager()`. In this function, we request the list of valid Subject Manager references from the local CORBA Service and present the information about each one to the user. Using this information, the user selects one, and that reference is then returned to the caller. In a real-world example, the Subject Manager chooser would be much more sophisticated than what we show in Example 7.9, "client.cpp: Implementation of `chooseSubjectManager()`", but for the purposes of explaining the ideas, this will suffice.

Example 7.9. client.cpp: Implementation of `chooseSubjectManager()`

```

1 tweek::SubjectManager_var
  chooseSubjectManager(tweek::CorbaService& corbaService)
  {
2     tweek::SubjectManager_var subj_mgr;
3
4     // Request all the active Subject Manager references.
5     std::list<tweek::SubjectManager_var> mgrs =
6         corbaService.getSubjectManagerList();
7
8     std::list<tweek::SubjectManager_var>::iterator cur_mgr;
9
10    // Iterate over all the tweek::SubjectManager references we
11    // have.
12    for ( cur_mgr = mgrs.begin();
13         cur_mgr != mgrs.end();
14         ++cur_mgr )
15    {
16        try
17        {
18            // It is not entirely safe to assume that *cur_mgr is
19            // still valid at this point, even though it was valid
20            // when the mgrs list was constructed. Hence, we test
21            // it again now.
22            if ( ! (*cur_mgr)->_non_existent() )
23            {
24                std::string response;
25                const std::string proceed("y");
26
27                tweek::SubjectManager::SubjectManagerInfoList_var
28                    mgr_info = (*cur_mgr)->getInfo();
29
30                std::cout << "\nSubject Manager information:"
31                    << std::endl;

```

1**2****3****4**

```

35         // Loop over the information items and print each
           // key/value pair to the screen.
           for ( CORBA::ULong i = 0;
                i < mgr_info->length();
                ++i )
40         {
           std::cout << "\t" << mgr_info[i].key << " = "
                     << mgr_info[i].value << std::endl;
           }

45         std::cout << "Use this Subject Manager (y/n)? ";
           std::cin >> response;

           if ( proceed == response )
           {
50             subj_mgr = *cur_mgr;
               break;
           }
       }
55     catch (...)
       {
           vprDEBUG(vprDBG_ALL, vprDBG_CRITICAL_LVL)
             << "Caught an unknown exception in "
             << "chooseSubjectManager loop\n"
60             << vprDEBUG_FLUSH;
       }
   }

   return subj_mgr;
65 }

```

5**6**

1 Using the given `tweek::CorbaService` reference, we ask for the current list of valid `tweek::SubjectManager` references. The references in this list are guaranteed to be valid at the time the list was constructed.

2 We use an STL iterator to loop over the list of returned `tweek::SubjectManager` references. This loop presents each reference in sequence and asks the user if the current reference is the one he wants. The loop completes when the user selects a reference or when no more references are available.

3 While the list of Subject Manager references was guaranteed to have contained valid references when it was constructed, things may have changed since then. That is the nature of asynchronous programming. To be safe, we test the current Subject Manager reference to see if it still refers to an extant object. If so, we continue. If not, we skip it. This invocation of `tweek::SubjectManager::_non_existent()` may throw an exception, and for that reason, we enclose the body of the for loop in a try/catch block.

5 It is possible for a single Naming Service to have multiple active Subject Manager references. Each of these has a unique identifier within the Naming Service, but the identifier is not human-readable. To work around this, we make use of the method `tweek::SubjectManager::getInfo()`. This method returns a sequence of key/value pairs (both are strings) that can be used to identify which Subject Manager reference is the correct one. Here, we request this information sequence and print all the key/value pairs. With this output, the user can (hopefully) determine which Subject Manager reference to use.

Note

This example is purposefully simple to keep the code small. Readers are strongly encouraged to come up with much more sophisticated choosing mechanisms here. For example, the flexibility of the Subject Manager information sequence would allow choices to be made entirely in code without any interactive user feedback.

6 Here, the user has selected the current Subject Manager as the correct one. We copy the reference to the `subj_mgr` variable. Then, we break out of the loop and return `subj_mgr` to the caller.

With that, we are done with our review of the C++ client API in Tweak. The use of CORBA allows Java and C++ client code to be quite similar, and this can be helpful when migrating between the two. The addition of the C++ client API in Tweak 0.13 also demonstrates one of the basic design philosophies of Tweak: clients can be written in any language without concern for the language the server uses.

Chapter 8. Python

The use of Python with Tweak is still being refined. While programmer's can make use of Python with Tweak right now, Tweak includes very little built-in support to ease the use of Python. Such features are planned for Tweak 1.2.

Part III. Appendices

Table of Contents

A. Compiling Example Code	56
SliderSubject	56
File Loader	58
B. CORBA Implementations	59
C. Legal	60
D. GNU Free Documentation License	61
PREAMBLE	61
APPLICABILITY AND DEFINITIONS	61
VERBATIM COPYING	62
COPYING IN QUANTITY	62
MODIFICATIONS	63
COMBINING DOCUMENTS	64
COLLECTIONS OF DOCUMENTS	65
AGGREGATION WITH INDEPENDENT WORKS	65
TRANSLATION	65
TERMINATION	65
FUTURE REVISIONS OF THIS LICENSE	66
ADDENDUM: How to use this License for your documents	66

Appendix A. Compiling Example Code

This appendix provides makefiles that can be used as starting points for compiling the example code shown earlier. These can also be used as the basis for future projects using Tweek.

Important

When compiling on Windows, special care must be taken to manage DLL symbol importing correctly. The following preprocessor symbols *must* be defined when compiling *all* user code:

- USE_core_stub_in_nt_dll
- USE_core_stub_in_nt_dll_NOT_DEFINED_Subject

It is especially important to define these two symbols when compiling C++ code generated by the omniORB IDL compiler (**omniidl**). Failure to define these symbols will result in run-time errors involving the registration of subjects with the Tweek Subject Manager. Other problems may arise as well.

The best way to define these symbols is to use the /D option to **CL.EXE** so that every C++ file compiled has the symbols defined. This can be done easily using a Visual C++ project file or in a makefile. Example Visual C++ project files can be found with the Tweek test applications under the directory tree %TWEЕК_BASE_DIR%\share\tweek\test.

SliderSubject

The following is a makefile that shows how to compile the code related to the SliderSubject example presented in the section called “Collaborative Slider”. It runs the IDL compiler, the C++ compiler, and the Java compiler. It assumes the use of **omniidl**, the JacORB compiler, and GCC in a Linux environment.

```
APP_NAME=      server

all:
    $(MAKE) cxx_idl
    $(MAKE) java_idl
    $(MAKE) NO_DEPEND=0 cxx
    $(MAKE) $(APP_NAME)
    $(MAKE) java
    $(MAKE) NetworkTestBean.jar
    -$(MAKE) install

# Basic options.
srcdir=      .
NO_DEPEND=   YES

IDL_CXX_FILES=  SliderSubject.cpp SliderSubject.h      \
                WhiteboardSubject.cpp WhiteboardSubject.h
IDL_JAVA_FILES= SliderSubject.java                    \
                WhiteboardSubject.java

base_srcs=     SliderSubjectImpl.cpp                  \
                WhiteboardSubjectImpl.cpp            \
                main.cpp

idl_srcs=      \
    $(filter-out $(base_srcs), $(notdir $(wildcard $(CXX_GEN_DIR)/*.cpp)))
```

```

SRCS=                $(base_srcs) $(idl_srcs)
JAVA_SRCS=           networktest/NetworkTest.java           \
                    networktest/SliderObserverImpl.java    \
                    networktest/SliderSubjectHolder.java    \
                    networktest/WhiteboardObserverImpl.java \
                    networktest/WhiteboardSubjectHolder.java
ALL_CLASSES=        networktest/*.class

DZR_BASE_DIR=       $(shell flagpoll doozer --get-prefix)
include $(DZR_BASE_DIR)/ext/tweek/dzr.tweek.mk

CXX_GEN_DIR=        .
CXX_IDL=             $(TWEED_IDL_CXX)
CXX_IDL_OPTS=       $(TWEED_IDLFLAGS_CXX) \
                    $(TWEED_IDL_GENDIR_CXX)$(CXX_GEN_DIR)
CXX_IDL_INCLUDES=
EXTRA_INCLUDES+=   -I$(CXX_GEN_DIR)

JAVA_GEN_DIR=       $(srcdir)
JAVA_IDL=           $(TWEED_IDL_JAVA)
JAVA_IDL_OPTS=     $(TWEED_IDLFLAGS_JAVA) \
                    $(TWEED_IDL_GENDIR_JAVA) $(JAVA_GEN_DIR)
JAVA_IDL_INCLUDES=

vpath %.idl $(srcdir)
vpath %.cpp $(CXX_GEN_DIR)
vpath %.h $(CXX_GEN_DIR)
vpath %.java $(JAVA_GEN_DIR)

# -----
# Application build targets.
# -----
cxx_idl: $(IDL_CXX_FILES)

%.cpp %.h: %.idl
    $(CXX_IDL) $(CXX_IDL_OPTS) $(CXX_IDL_INCLUDES) $<

java_idl: $(IDL_JAVA_FILES)

%.java: %.idl
    $(JAVA_IDL) $(JAVA_IDL_OPTS) $(JAVA_IDL_INCLUDES) $<

cxx: $(OBJS)

java: $(CLASSES)

server: $(OBJS)
    $(LINK) $(LINK_OUT)$@ $(OBJS) $(TWEED_LIBS) \
    $(EXTRA_LIBS) $(LIBS)

NetworkTestBean.jar: $(JAVA_CLASSES)
    $(JAR) cvf $@ $(ALL_CLASSES)

install:
    cp NetworkTestBean.jar $(TWEED_BASE_DIR)/share/tweek/beans
    cp $(srcdir)/NetworkTestBean.xml \
    $(TWEED_BASE_DIR)/share/tweek/beans

CLEAN_FILES+= \
    $(addprefix $(srcdir)/, $(IDL_CXX_FILES)) \
    $(addprefix $(srcdir)/networktest/, $(IDL_JAVA_FILES)) \
    $(CLASSES) tweek/*.class
CLEAN_DIRS+= $(srcdir)/tweek

```

```
CLOBBER_FILES+= NetworkTestBean.jar
```

File Loader

The following is a makefile that shows how to compile the code related to the file loader Bean example presented in the section called “File Loader”. All it must do is compile the Java code for the Bean.

```
all:
    $(MAKE) java
    $(MAKE) FileOpenTestBean.jar
    -$(MAKE) install

# Basic options.
srcdir=      .
NO_DEPEND=   YES

JAVA_SRCS=   fileopentestbean/FileOpenTestBean.java
ALL_CLASSES= fileopentestbean/*.class

DZR_BASE_DIR= $(shell flagpoll doozer --get-prefix)
include $(DZR_BASE_DIR)/ext/tweek/dzr.tweek.mk

# -----
# Application build targets.
# -----
java: $(CLASSES)

FileOpenTestBean.jar: $(JAVA_CLASSES)
    $(JAR) cvf $@ $(ALL_CLASSES)

install:
    cp FileOpenTestBean.jar $(TWEAK_BASE_DIR)/share/tweek/beans
    cp $(srcdir)/FileOpenTestBean.xml \
        $(TWEAK_BASE_DIR)/share/tweek/beans

CLOBBER_FILES+= FileOpenTestBean.jar
```

Appendix B. CORBA Implementations

In order to use CORBA, a CORBA implementation must be available. As of this writing, Tweek uses omniORB 3.0.4 and newer [Omn02] for a C++ ORB. Since Tweek uses the POA, any C++ ORB may be used with only a few changes to the code.

For the Java side, OpenORB 1.2.0 [Ope02] and newer [Ope03] have been used. Java IDL [Jid02], the CORBA implementation that comes with the Java Development Kit (JDK), does not fully implement the POA as of JDK 1.3.1. With the release of JDK 1.4.0, Java IDL has a complete, working POA implementation and can be used without taking any special steps.

To use an alternate ORB with Java, two arguments must be passed to the Java virtual machine. They are based on the ORB implementation. For example, if using OpenORB, the following two arguments must be specified:

1. `-Dorg.omg.CORBA.ORBClass=org.openorb.CORBA.ORB`
2. `-Dorg.omg.CORBA.ORBSingletonClass=org.openorb.CORBA.ORBSingleton`

Other ORBs will vary in the location of the `ORBClass` and the `ORBSingletonClass`. Refer to the documentation of the specific ORB to find out more about using it instead of Java IDL.

For Python, most testing thus far has been done with omniORBpy 2.0 and newer. Using omniORBpy requires that omniORB be installed since omniORBpy is primarily just a Python wrapper around the C++ CORBA implementation. We are watching the development of Fnorb [Fno03] as an alternative to omniORBpy. Fnorb offers a pure-Python implementation of CORBA. In other words, it requires no natively compiled code to operate. Prior to December 2003, Fnorb lacked a POA implementation, so it was not a viable option, but work is still under way to provide a complete CORBA 2.3 implementation.

Appendix C. Legal

A full binary distribution of Tweek comes with binary code from several other software projects not affiliated with the Juggler Project or Iowa State University. Tweek makes use of redistributable code from the following packages:

- JacORB (<http://www.jacorb.org/>)
- JDOM (<http://www.jdom.org/>)
- JGoodies look and feel (<http://www.jgoodies.com/>)
- Kunststoff look and feel (<http://www.incors.org/>)
- Liquid look and feel (<http://liquidlnf.sourceforge.net/>)
- Metouia look and feel (<http://mlf.sourceforge.net/>)
- OpenORB (<http://openorb.sourceforge.net/>)

The licenses for each of these software packages can be found in `$TWEЕК_BASE_DIR/share/tweek/data/licenses`.

Appendix D. GNU Free Documentation License

Version 1.2, November 2002

FSF Copyright note

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sec-

tions then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you

as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

GNU FDL Modification Conditions

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the

title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the

combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Sample Invariant Sections list

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

Sample Invariant Sections list

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

- [Fno03] Fnorb website [<http://www.fnorb.org/>].
- [Gam95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Copyright © 1995 Addison Wesley Longman, Inc.. 0-201-63361-2. Addison-Wesley.
- [Hen99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Copyright © 1999 Addison Wesley Longman, Inc.. 0-201-37927-9. Addison Wesley Longman, Inc..
- [Jbe02] JavaBeans website [<http://java.sun.com/products/javabeans/>].
- [Jid02] Java IDL website [<http://java.sun.com/products/jdk/idl/>].
- [Omn02] omniORB website [<http://www.uk.research.att.com/omniORB/>].
- [Ope02] OpenORB website [<http://www.openorb.com/>].
- [Ope03] OpenORB community website [<http://openorb.sourceforge.net/>].
- [Pyq03] PyQt website [<http://www.riverbankcomputing.co.uk/pyqt/index.php>].

Glossary

A

application programmer interface

An application programmer interface is the documented, programmatic interface used by programmers to access a software library.

C

Common Object Request Broker Architecture

The Common Object Request Broker Architecture (CORBA) is a standard specified by the Object Management Group for distributed object-oriented programming. It is both platform- and language-independent. Implementations of CORBA are available from many vendors for a wide variety of programming languages.

D

distributed programming

Distributed programming is a paradigm in which software components are installed on physically separated computers and accessed remotely through a network. Examples of distributed programming include Sun's Remote Procedure Calls (RPC), Java's Remote Method Invocation (RMI), Microsoft's Component Object Model (COM), and the Common Object Request Broker Architecture (CORBA). RPC is based on the procedural programming paradigm used by languages such as C and Pascal. With RPC, procedures residing in memory on remote machines are accessed. The other examples listed are based on the object-oriented programming paradigm used by languages such as C++, Java, Smalltalk, and Python. In this case, software objects and their methods are accessed remotely.

E

Extensible Markup Language

The eXtensible Markup Language (XML) is a specification from the World Wide Web Consortium. XML itself is a language for specifying well-defined, structured document markup.

G

graphical user interface

Graphical user interfaces are used in modern operating systems that support the concept of "windows". The windows provide a visual representation of an application and usually have a border and a title to distinguish one window from another. Within the window, there are graphical elements known as "widgets" that make up the full interface. Widgets include pull-down menus, clickable buttons, and scroll bars.

I

IDL compiler	A software tool that reads an IDL file and generates code in a specific language (e.g., Java, C++, Smalltalk, etc.). This code may be anything, but typically, it is stub or skeleton code that is extended by user-defined code that completes the interface implementation.
interface	Several types of “interfaces” exist within the scope of Tweek: <ul style="list-style-type: none">• In Java, there is a keyword <code>interface</code> that can be used to define a set of methods that must be part of an implementing class. Classes that claim to implement a given interface must define the methods described by the interface or declare themselves abstract.• In <i>IDL</i>, an interface is essentially identical to a Java interface, but an IDL compiler can generate code from the IDL. The interface methods must still be implemented (by a C++ class, for example), and thus, the main function of an IDL interface is to define a language-independent signature through which CORBA communication may occur.• In C++, there is no <code>interface</code> keyword as in Java and IDL. The term “interface” is typically applied to abstract classes (those with pure virtual methods). More generally, it is used to refer to the collection of methods defined by a class.
Interface Definition Language	The Interface Definition Language is a simple language used to define the interface (or signature) an object will have. An interface is composed solely of methods (functions) that may be invoked on an object (data element). No data members are present in the interface definition. IDL is not tied to a specific language but instead must be compiled into another language where the interface will be implemented.
Internet Inter-ORB Protocol	The Internet Inter-ORB Protocol (IIOP) is part of the CORBA specification. It is used to standardize communication between ORBs so that ORBs from different vendors can inter-operate seamlessly.

J

JavaBean	JavaBeans are the component architecture of the Java programming language. There are no restrictions on the interfaces implemented by JavaBeans, though it is strongly recommended that they implement <code>java.io.Serializable</code> .
Java Native Interface	The Java Native Interface is the bridge between a Java virtual machine (JVM) and natively compiled code. Native code can load a JVM and get access to Java objects through it, or Java classes may have native methods that are loaded by the JVM.

O

Object Request Broker An Object Request Broker (ORB) is used by CORBA for management of interface implementation objects (servants) and the references made to those objects. ORBs communicate with each other using a client/server model, though two ORBs may both act as servers and clients during the transactions.

P

Portable Object Adapter The Portable Object Adapter (POA) is the standard object adapter and is part of the CORBA 2.3 specification. It is used together with IIOP to allow ORB implementations from different vendors to communicate. A single ORB may have multiple POAs each tailored to a certain task using parameterized characteristics.

R

Remote Method Invocation Remote Method Invocation (RMI) is a mechanism for gaining access to remote Java code. It is implemented as part of the Java virtual machine and was originally intended to be comparable to CORBA. RMI allows easy access to code between two virtual machines, but it does not have the cross-language features of CORBA.

S

servant In CORBA, a servant is an instance of an interface implementation that is registered with an ORB. A servant receives method invocations through the CORBA object adapter.

X

XML See *Extensible Markup Language*.

Index

B

Bean descriptions
using XML, 8

C

C++ client API, 41
C++ server API, 13
classes
 POA_tweek::Observer, 4
 tweek.ObserverPOA, 4, 26
 tweek::SubjectImpl, 3, 3
CORBA
 use in Tweek, 4
CORBA Manager, 3
 design and use, 4
 initialization
 example of, 28
 use of, 15

G

Generic Beans, 8

I

IDL, 3, 5
interfaces
 tweek::Observer, 3
 tweek::Subject, 3, 11, 13

J

Java client API, 21
Java libraries
 Bean, 21
 Bean Delivery, 22
 Event, 21
 GUI, 21
 Network, 21

O

observer
 design and use, 3
org.vrjuggler.tweek.net.CorbaService, 41

P

Panel Beans, 8
Python client API, 53

S

Service Beans, 7
subject
 design and use, 3
Subject Manager, 3

creation
 example of, 28
design and use, 4
use of, 17

T

tweek::CorbaService, 41, 41
tweek::Subject
 IDL definition, 11
tweek::SubjectImpl
 deriving from, 13
tweek::SubjectManagerImpl
 addInfoItem() method, 20
 setApplicationName() method, 19
 setUserName() method, 20

V

Viewer Beans, 7

X

XML, 8