

University of Nevada
Reno

A Unified Approach for Cross-Platform Software Development

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
with a major in Computer Science.

by

Jeffery Alan Stuart

Dr. Frederick C. Harris, Jr., Thesis advisor

August 2005

UNIVERSITY
OF NEVADA
RENO

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

JEFFERY ALAN STUART

entitled


**A Unified Approach for Cross-Platform Software
Development**

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE



Frederick C. Harris, Jr. Ph.D., Advisor



Sergiu Dascalu, Ph.D., Committee Member



Jerry Johnson, Ph.D., Graduate School Representative



Marsha H. Read, Ph. D., Associate Dean, Graduate School

August, 2005

Abstract

Cross-platform software development is a complex and challenging activity. Frequently, developers have to create portions of code that use platform-specific data types and functions. This has led to two largely adopted practices: either making extensive use of the preprocessor, or splitting the software package into several branches, one for each target platform. Both practices have their drawbacks. To tackle the issues of cross-platform development, this thesis proposes two programming solutions referred to as cores and routers. By using them, the need for advanced preprocessing and separate development branches is virtually eliminated. The conceptual solutions are described and examples of application are presented.

Contents

Abstract	i
List of Figures	iii
List of Tables	iv
1 Introduction	1
2 Cross-Platform Software Development	3
2.1 History of Cross-Platform Development	3
2.2 Preprocessor Use	4
2.2.1 Example 1: Boost Threads	5
2.2.2 Example 2: HawkNL	5
2.2.3 Example 3: MsgConnect	6
2.2.4 Example 4: ZThreads	8
2.2.5 Comparison of Packages	8
2.3 Separate Development Branches	9
2.4 A Taxonomy of Software Packages	10
2.5 Design Patterns	11
3 Cores and Routers	13
3.1 Introduction	13
3.2 Overview of Cores and Routers	14
3.3 Core Example	16
3.4 Router Example	20
4 The JTK and PCM-Client/Server	25
4.1 JTK	25
4.1.1 Package: jtk	26
4.1.2 Package: jtk::io	27
4.1.3 Package: jtk::media	28
4.1.4 Package: jtk::net	29
4.1.5 Package: jtk::os	30
4.1.6 Package: jtk::util	30

4.2	PCM-Client/Server	32
4.2.1	PCM-Server	33
4.2.2	PCM-Client	34
5	Future Work	36
6	Conclusions	38

List of Figures

2.1	Excerpt from Boost Threads Source Code (thread.hpp)	5
2.2	Excerpt from HawkNL Source Code (nl.h)	6
2.3	Excerpt from MsgConnect Source Code (MC.h)	7
2.4	Excerpt from ZThreads Source Code (ThreadOpts.h)	8
2.5	Excerpt from Qt Source Code (QWidget.h)	10
3.1	UML Diagram for Core solution	14
3.2	UML Diagram for Router solution	15
3.3	Non-Core Thread implementation	18
3.4	Thread implementation	19
3.5	ThreadCore implementation	19
3.6	Win32ThreadCore implementation	20
3.7	PosixThreadCore implementation	21
3.8	UML Diagram for Thread core	21
3.9	Non-System File implementation	22
3.10	File implementation	23
3.11	FileSystem implementation	23
3.12	Posix File System implementation	24
3.13	UML Diagram for File router	24
4.1	The JTK Package hierarchy	26
4.2	The jtk package	27
4.3	The jtk::io package	28
4.4	The jtk::media package	29
4.5	The jtk::net package	30
4.6	The jtk::os package	31
4.7	The jtk::util package	32
4.8	PCM-Server Flow Diagram	33
4.9	PCM-Client Flow Diagram	34

List of Tables

2.1	Comparison of Preprocessor Based Software Packages	9
2.2	Taxonomy of Software Libraries	11

Chapter 1

Introduction

Cross-platform software development is an intricate and demanding activity [2, 4, 10, 15, 17, 18, 26]. Very often, the developers have to create parts of code that require platform-specific data types and functions. This requirement has led to two largely adopted practices: either making extensive use of the preprocessor, or dividing the software package into several segments (branches), each corresponding to a target platform [7]. The former practice frequently leads to unreadable code, making later modifications difficult and error-prone. The latter approach can lead to lack of organization and code not being shared across branches. These drawbacks were experienced in various software projects such as [8, 16, 27].

Starting from these observations, this thesis examines several cross-platform software packages and identifies common traits between the packages. Then, using fundamental object-oriented programming techniques, two design and implementation solutions referred to as cores and routers are used to address the issues of cross-platform software development in object-oriented programming languages. A core provides the underlying data types and operations necessary for platform-specific code whereas a router provides only the needed operations (more precisely, a router is used when platform-independent data types can adequately represent the state of an object). By using cores and routers, the need for advanced preprocessing and/or separate development branches is practically eliminated. The code produced is much more readable, while the absence of separate platform-dependent development branches allows for more efficient code sharing.

Based on cores and routers, two simple yet effective (and rather symmetrical) development solutions, a more consistent (“unified”) approach for cross-platform software development is suggested. The idea is implemented in C++, though other object-oriented languages can be used. Although the idea of the proposed approach might seem obvious at the first sight, research and study show that it has not yet been applied – at least not on a larger scale, for example in popular packages for cross-platform software development such as [14], [22], and [23]. The proposed cross-platform software development core and router solutions are illustrated in this thesis through several examples of application. Details about the intended meaning of the specific terms used (cores and routers), are also provided.

The remainder of this thesis is organized as follows: Chapter 2 presents background information on current cross-platform software development practices, design patterns, and related issues, Chapter 3 introduces the concepts of cores and routers intended to help solve these issues, as well as an example implementation of a core and an example implementation of a router. Chapter 4 presents a cross-platform library called the **JTK** developed using cores and routers, as well as an application made using the **JTK**. Chapter 5 outlines several directions of future work, and Chapter 6 concludes with a summary of the thesis’s contributions.

Chapter 2

Cross-Platform Software Development

This chapter discusses the common methods employed for cross-platform software development. Section 2.1 discusses the history and gives a brief timeline of cross-platform development methods. Section 2.2 discusses the first method of cross-platform software development: making heavy use of the preprocessor to strip source code from inclusion at compile time. Section 2.3 discusses the second method of cross-platform software development: using separate development branches for each targeted platform. Section 2.4 presents a taxonomy of the software packages surveyed by this thesis, while Section 2.5 gives a brief overview of design patterns and their significance with respect to cross-platform software development.

2.1 History of Cross-Platform Development

Perhaps the best way to understand why cross-platform development techniques is by examining how and why they originated. For a large part of early UNIX development [25], ranging from approximately 1969 to 1976, there was no such thing as cross-platform software. Software was written for one very specific hardware/software platform. If the platform became extinct, the entire package would be ported to the old platform's successor.

Once multiple hardware and software platforms became popular, the UNIX kernel needed a way to compile and run on many different platforms. With the advent of

the program **make** and **Makefiles** [12] in 1979, the preprocessor became widely used in applications that sought to compile on multiple platforms. Originally, the makefile was made not only to smart-compile source files, but also to pass excessively long and numerous command line arguments to the compiler. The majority of the arguments passed were preprocessor defines.

As time progressed, more software packages were targeted for multiple platforms, and eventually, the idea of separate development branches, one for each platform, came about. Instead of having source files contain code for every platform, development would be split into branches. Each branch contained only the source code necessary for a specific platform, thereby reducing the number of conditional checks by the preprocessor.

2.2 Preprocessor Use

In the first approach for cross-platform software development, source files are sprinkled with preprocessor statements, which makes the code hard to read, error-prone, difficult to test on all platforms, and hard to maintain. Examples of software packages that rely on this approach are Boost Threads [3], HawkNL [21], MsgConnect [5], and ZThreads [6]. Also, one might note that the POSIX [22] Threads package employs both of the methods mentioned, but this thesis considers it a stronger fit for the “separate branches” category.

An interesting observation is that even though all of these packages make heavy use of the preprocessor, they all use it in different ways, as is described below in their respective subsections. However, even with four different methods of using the preprocessor, no greater readability or code security is achieved. The methods employed by each package are most likely used because the respective authors of each package have grown accustomed to said methods. While this thesis in no way seeks to proclaim one of these packages superior to any other, it does make certain observations about the safety and readability of each method.

2.2.1 Example 1: Boost Threads

Boost Threads [3] is a cross-platform threading API written in C++ with the goal of being adopted into standard C++ library specifications and implementations. Boost has possibly the most dangerous and hardest to read methods of using the preprocessor. Boost uses the preprocessor to section out platform-specific function implementations. Boost also uses the preprocessor to conditionally determine what variables are to be stored within a class. This practice can be extremely dangerous as one might expect a certain preprocessor condition to be true, when in fact it is not, making the class's new definition incompatible with the binary representation and possibly causing a highly untraceable runtime error. An example of this method, part of the **Boost Threads** source code, is shown in Figure 2.1.

```
class BOOST_THREAD_DECL thread : private noncopyable
{
public:
    ... source code is here

private:
#ifdef BOOST_HAS_WINTHREADS
    void* m_thread;
    unsigned int m_id;
#elif defined(BOOST_HAS_PTHREADS)
private:
    pthread_t m_thread;
#elif defined(BOOST_HAS_MPTASKS)
    MPQueueID m_pJoinQueueID;
    MPTaskID m_pTaskID;
#endif
    bool m_joinable;
};
```

Figure 2.1: Excerpt from Boost Threads Source Code (thread.hpp)

2.2.2 Example 2: HawkNL

HawkNL [21] is a cross-platform networking API written in C. HawkNL uses the preprocessor in a similar manner to Boost Threads. However, instead of declaring members of a class or structure conditionally, HawkNL uses conditional typedefs and preprocessor defines. This practice leads to safer use of code, but is still confusing to

the average programmer, especially when one tries to read through header and source files. An example of HawkNL’s use of the preprocessor is shown in Figure 2.2.

```

/* Any more needed here? */
#if defined WIN32 || defined WIN64 || defined __i386__ || \
    defined __alpha__ || defined __mips__
#define NL_LITTLE_ENDIAN
#else
#define NL_BIG_ENDIAN
#endif

/* How do we detect Solaris 64 and Linux 64 bit? */
#if defined WIN64
#define IS_64_BIT
#endif

/* 8 bit */
typedef char NLbyte;
typedef unsigned char NLubyte;
typedef unsigned char NLboolean;
/* 16 bit */
typedef short NLshort;
typedef unsigned short NLushort;
/* 32 bit */
typedef float NLfloat;
#ifdef IS_64_BIT
typedef int NLlong;                /* Longs are 64 bit on a 64 bit
                                   CPU, but integers are still
                                   32 bit. */
typedef unsigned int NLulong;     /* This is, of course, not true
                                   on Windows (yet another
                                   exception), */
                                   /* but it does not hurt. */
#else
typedef long NLlong;
typedef unsigned long NLulong;
#endif

```

Figure 2.2: Excerpt from HawkNL Source Code (nl.h)

2.2.3 Example 3: MsgConnect

MsgConnect [5] is a cross-platform networking API written in C++. Its use of the preprocessor is slightly different from other packages. It has a few “include” files that create preprocessor macros based on other preprocessor defines. An example is that in order to put a thread to sleep, MsgConnect makes a macro called **Sleep(x)**. If MsgConnect is compiled for windows, the preprocessor is told to define the **Sleep(x)** macro as a call to the Win32 function **Sleep**. If, however, MsgConnect is compiled for a Linux platform, the preprocessor is told to define the **Sleep(x)** macro as a call

to the standard **usleep** function.

As can be seen in an excerpt from `MsgConnect` shown in Figure 2.3, many macros are being defined as their Win32 equivalents. Many constants are also being defined to Win32 constants if the platform being targeted is not Windows. This can be somewhat confusing to programmers with experience in both Win32 and UNIX/Linux programming. They can (and the author of this thesis has done this before) look through the source code and think they do not have the right version, because they were expecting code for Linux and what appears in front of them seems to be Win32 API calls and Win32 constants. The same scenario would hold if a package tried to define as many POSIX (or Linux) constants and macros, and a programmer was intending to build something for a Win32 target. This method of using the preprocessor most likely came as a side-effect of the package's development schedule. The package was probably made for Windows initially, and in an attempt to minimize the cost of porting to other platforms the author(s) decided to emulate the Win32 API in Linux by using macros and typedefs with conditional compiling. Much of the `MsgConnect` code is written this way.

```

#ifdef __GNUC__
#ifndef __USE_UNIX98
#   define __USE_UNIX98
#endif

typedef long long __int64;
typedef unsigned char byte;
const unsigned long INFINITE = 0xFFFFFFFFul; // Infinite timeout
const unsigned long INVALID_SOCKET = (unsigned long)-1;
const unsigned long SOCKET_ERROR = (unsigned long)-1;
unsigned long GetTickCount(void);

# define TCHAR char
# define __T(x)      x
# define _T(x)      __T(x)
# define TEXT(x)    __T(x)
# define _TEXT(x)   __T(x)
# define LPTSTR  char *
# define LPCTSTR const char *
#   define Sleep(s) usleep((long)s*1000)
// more preprocessor defines are here
#else
#   include <windows.h>
#endif

```

Figure 2.3: Excerpt from `MsgConnect` Source Code (`MC.h`)

2.2.4 Example 4: ZThreads

ZThreads [6] is a cross-platform threading API written in C++. The use of preprocessor statements within contains the normal conditional defines and execution statements, but also performs a clever trick with **include** statements. In some places, instead of wrapping sections of executable code in preprocessor statements, ZThreads will instead include platform-specific implementation source code. This aids in efficiency of the code, but still makes tracing through source files a problem as it can be hard to determine which files are being included without digging through the entire library. Most of the ZThreads code is in written this manner. Figure 2.4 shows an example of ZThreads conditionally defining which implementation files to include.

```

#if defined(ZT_POSIX)
#   include "posix/ThreadOps.h"
#   define ZT_THREADOPS_IMPLEMENTATION "posix/ThreadOps.cxx"
#elif defined(ZT_WIN32) || defined(ZT_WIN9X)
... more source code here
#   include "win32/ThreadOps.h"
#   define ZT_THREADOPS_IMPLEMENTATION "win32/ThreadOps.cxx"
#elif defined(ZT_MACOS)
#   include "macos/ThreadOps.h"
#   define ZT_THREADOPS_IMPLEMENTATION "macos/ThreadOps.cxx"
#endif

```

Figure 2.4: Excerpt from ZThreads Source Code (ThreadOps.h)

2.2.5 Comparison of Packages

All four packages described previously use preprocessor statements to conditionally compile code. Table 2.1 gives a summation of the side effects of each package's use of the preprocessor.

Table 2.1: Comparison of Preprocessor Based Software Packages

	Boost Threads	HawkNL	MsgConnect	ZThreads
Conditionals in functions	X			
Variable size structures	X	X	X	X
Conditional in source	X	X		X
“Emulated” platform			X	

2.3 Separate Development Branches

The second approach for cross-platform software development is categorized as having source code split into separate development branches. Most of the packages this thesis identifies as belonging to this category are quite large and continue to be in a state of active development. The use of this development method seems to be common, and is most likely attributed to the vast amount of resources and knowledge (of more than one platform) that the programmers for these projects have.

Examples of projects using separate development branches are OpenSG [19], Qt [23], GTK [14], and POSIX Threads [22]. All these projects are still in active development and/or refinement, and they all support many platforms. For example, OpenSG supports four operating systems (Windows, Vanilla Linux, Irix, and Irix 64), with a total of five compilers (Intel’s Compiler, Microsoft’s Optimizing Compiler for ISO C++, GCC 3.2 and above, SGI CC, and SGI 64 bit CC).

Several unfortunate side effects are often seen with a package being split into several development branches. First and foremost, code sharing is minimized. Sometimes access to platform-specific objects and functionality is granted, thus making it harder to write platform-independent code using the package. Perhaps the most severe side-effect (in terms of development) is the case when a small design change or enhancement is made, and that change requires different levels of effort to incorporate on the many different platforms.

Using separate development branches, code might be written for one platform that could simply be “dropped-in” with another development branch. However, code-sharing usually does not happen because the programmers of the different branches

tend to be isolated from each other.

In some packages, an application programmer is given access to platform-specific functionality in such a manner as to break the write-once/compile-anywhere model. Qt allows this in some cases, and a code excerpt from the base class for all of Qt's windowing gadgets, **QWidget**, is shown in Figure 2.5. Since the QWidget class gives access to its platform-specific functionality in a trivial manner, a naive application programmer might be tempted to use the platform-specific functionality without realizing the consequences.

```

#ifndef QWIDGET_H
#define QWIDGET_H

class Q_EXPORT QWidget : public QObject, public QPaintDevice
{
    ... more source is here
public:
    QWidget( QWidget *parent=0, const char *name=0, WFlags f=0 );
    ~QWidget();

    WId    winId() const;
    void  setName( const char *name );
    ... more source is here
};

```

Figure 2.5: Excerpt from Qt Source Code (QWidget.h)

From the library development point of view, managing multiple branches can be a daunting task. A small change (simple addition or subtraction of a member function) in one branch might instigate an entire code re-write in another branch. This forces the API and library designers to have an intricate knowledge of how each branch is implemented, so that when a change is proposed, one development branch doesn't suddenly fall behind all the other branches.

2.4 A Taxonomy of Software Packages

A taxonomy of the above software packages is shown in Table 2.2. One idea to be inferred from this categorization is that larger software packages tend to use separate development branches, most likely because there is a larger collection of developers

to assist in all the different branches of code. With smaller packages, typically there is only a handful of primary authors and, with such, breaking the work into several branches is an unnecessary complication. This segregation of programming methods is undesirable and is a problem this thesis addresses.

Table 2.2: Taxonomy of Software Libraries

Heavy Preprocessing	Separate Development Branches
ZThreads	Qt
MsgConnect	GTK
HawkNL	POSIX
Boost	

2.5 Design Patterns

Design patterns [9, 13] are representation of real-world programming solutions. Design patterns began to appear in published work in the late 1970s [1], as something more general; simply a better way to present ideas about planning, building, architecture, and representation. Design patterns allow programmers to easily share their program’s architecture, and can be split into three main categories [9]:

- Creational patterns
- Structural patterns
- Behavioral patterns

Creational patterns are used to help instantiate classes in non-standard ways (e.g. not explicitly calling a constructor). Structural patterns are used to make the design of complex classes and inheritance chains more elegant. Behavioral patterns define the ways in which classes interact with each other and in what ways a class may be used.

The two new solutions proposed by this thesis rely on creational patterns and structural patterns to provide a more elegant means of cross-platform API develop-

ment. Specifically, the design patterns **Bridge**, **Builder**, and **Abstract Factory** are used. Note that cores and routers are not identical to these patterns, though.

Chapter 3

Cores and Routers

Chapter 3 presents the new development solutions presented in this thesis. Section 3.1 provides the motivation behind this thesis' idea. Section 3.2 gives an overview of the solutions, referred to as cores and routers. Sections 3.3 provides details and an example of a core, and Section 3.4 provides details and an example of a router.

3.1 Introduction

Starting from the observations presented in Chapter 2 and driven by practical software development needs, a better method of writing cross-platform code was sought after. This method would have to address the seemingly hard to satisfy (at the same time) properties: readability and structure (on the one hand) and code sharing (on the other hand). In other words, the method would have to answer the question “how could the readability of the code be improved and the code sharing maximized while keeping the software organized?” The solution found is based on two simple, yet efficient development solutions, referred to as cores and routers. These are in fact two design constructs aimed at object-oriented implementation. From the experience gained in writing code for a general-purpose C++ APIs and from using it in actual software development projects it is shown that the resulting programs are easier to read and understand while at the same time the amount of code common for the platforms considered is maximized. The simplicity of the solutions (they are indeed meant to be easy to implement) and their rather symmetrical structure (which aids

quicker learning and memorization), allows a more consistent, smoother and “unified” way of writing cross-platform object-oriented library code. The concepts of *cores* and *routers* are introduced next with examples of their use.

3.2 Overview of Cores and Routers

A *core* is a generic code development solution that can be represented using the UML notation [24] as shown in Figure 3.1. In this figure a class (**Class A**) intended for cross-platform development relies on the services of the abstract class **Class A Core** which, in turn, has specialized platform-dependent implementations in the subclasses **Platform 1 Class A Core**, **Platform 2 Class A Core**, etc. Thus, a clean separation of platform-independent services from platform-dependent implementations is achieved. Because, generally speaking, most of the base (foundation) code for all cross-platform software is to be included in cores, the name core was used to highlight its pragmatic significance.

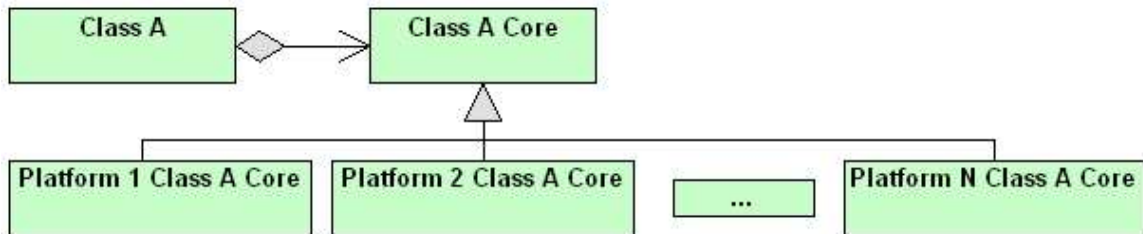


Figure 3.1: UML Diagram for Core solution

Note also that, in practical terms, our proposed generic core solution consists of a wrapper (principal) class, an abstract core class (for platform-independent services), and a set of concrete core classes (which implement the services provided in platform-dependent ways). An example of using the core solution is presented in Section 3.3 of this thesis.

The other component of our proposed approach for cross-platform software development is the router. A router is a generic code development solution, shown in Figure 3.2 using the UML notation. In this figure, **Class B** intended for cross-

platform software development relies on the services of the concrete class **Class B Router** which, in turn, is associated with the platform-dependent implementation classes **Platform 1 Class B Router**, **Platform 2 Class B Router**, etc. In contrast to the core solution, which relies on aggregation and inheritance relationships between its component classes, the router solution is based on dependency (“use”) relationships between classes.

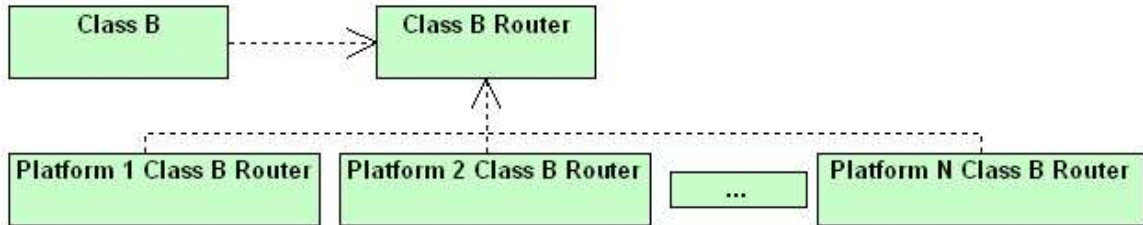


Figure 3.2: UML Diagram for Router solution

Note also that, in practical terms, our proposed generic router solution consists of a wrapper [13] (principal) class, a (main) router class and a set of (base) router classes (which implement services in platform-dependent ways). Regarding the terminology used, the name router was chosen because its design emphasizes the “routing” of required services towards appropriate implementations (from the wrapper and the base routers to the main router).

Technically speaking, the main router class could be eliminated from the design presented in Figure 3.2 by using several macros, but this would negatively (and significantly) affect the consistency of the programming style and the convenience of the existing regular source code structure (similar across the entire project in terms of associations among .h and .cpp files). Also, as shown later in this thesis, actual base router classes may not always be necessary.

Cores and routers borrow concepts from the both the Bridge and Factory [13] design patterns. Cores use the Factory design pattern to instantiate implementation classes. Both *cores* and *routers* rely on the Bridge design pattern to relay platform-dependent requests to platform-specific implementations. However, the use of *cores*

and *routers* extends beyond that of the Bridge, Factory, or any other previously defined concept or design pattern. However, one should note that neither a *core* nor a *router* are previously identified design-patterns/programming-solutions.

An example of using the router solution is provided in Section 3.4 of this thesis. To summarize the key concepts introduced, the rules for using cores and routers are the following:

- A core solution should be used when both (some of) the member variables and (some of) the operations of a class are platform-dependent;
- A router solution should be used when all the member variables of a class are platform-independent but (some of) the class operations are platform-dependent.

3.3 Core Example

A simple yet illustrative example for using cores is a Thread class. Threads are essential elements to practically every operating system, but there is no standard thread package so far (although the POSIX committee has tried to initiate the creation of one). Most UNIX platforms have their own threading library, virtually all flavors of Linux use POSIX threads (pthreads), and Microsoft Windows currently uses Win32 threads. All of these threads use platform-specific data types, for example POSIX threads use a `pthread_t` while Win32 threads use a `HANDLE`.

The expected functions for a Thread class are outlined as follows. First, a constructor and (because the implementation language chosen was C++) a destructor are needed. A function to start the execution of the thread should be available, so we include in this class a function called `start`. One might wish to wait for a thread to finish, so a `waitFor` function will be needed. One might also wish to relinquish the remaining time slice of a thread to the operating system, so a `yield` function is necessary. Of course, a thread is used to execute code, so an abstract function called `run` will be available for a user defined class to override. Obviously, more functionality is expected of a thread, but the above functionality is sufficient to demonstrate

the application of the proposed core concept.

Shown in Figure 3.3 is a previous implementation of the **Thread** class. The specific implementation used the preprocessor method to detect what platform is being used. Practically every other line is a preprocessor statement, and makes the reading of the code quite challenging. This is where the use of cores comes in. As explained in Section 3.2, a wrapper class, an abstract core class, and implemented core classes are needed. The implementation for the **Thread** wrapper class is shown in Figure 3.4. As it can be noticed, the **Thread** class in the core solution is merely an empty shell, relying on its core class for virtually all functionality. Also, one could notice that the only member variable is typically a core, and the **Thread** class is no exception.

The code for the abstract **ThreadCore** class is shown in Figure 3.5. The platform-independent functionality has been implemented but, as can be seen, all of the platform specific code is purposefully left out so child classes can implement the necessary functionality in a platform-specific manner. Next, shown in Figure 3.6 is the concrete **Win32ThreadCore** class, which implements all the remaining (platform-specific) code necessary to use a thread. Figure 3.7 shows the concrete **PosixThreadCore** class. Similar to the **Win32ThreadCore** class, this class implements all the platform-specific code necessary to use a thread.

While it may seem that more work is needed to use the core method than to employ the method exemplified in Figure 3.2, one must view the development process from a larger perspective. For example, any support for new platforms requires the modification of the **Thread** implementation file. This can become complicated when advanced users and developers make fine-tuned adjustments to their specific platform and then want to add support for a new platform. They cannot simply use the most up-to-date file provided by the **Thread** class maintainer, they must acquire the new **Thread** implementation and then manually merge the file with the changes they made. On the other hand, if one decides to use cores, any new platform support requires the modification of at most one file: the Makefile. If the users decide to


```

#ifdef _POSIX
#include <pthread.h>
#include <sched.h>
#elif defined(_WIN32)
#include <windows.h>
#endif
// Class representing an operating system execution thread
class Thread {
protected:
#ifdef _POSIX
pthread_t pthreadObject;
#elif defined(_WIN32)
DWORD threadID;
HANDLE threadHandle;
#endif
public:
Thread() { }
virtual ~Thread() { }
#ifdef _POSIX
void * threadStart(void * p) { ((Thread*)p)->run(); }
#elif defined(_WIN32)
DWORD WINAPI threadStart(void * p) { ((Thread*)p)->run(); }
#endif

// This function will halt until the passed in thread finishes
// its execution
static void waitFor(Thread * thread) {
#ifdef _POSIX
pthread_join(thread->pthreadObject, NULL);
#elif defined(_WIN32)
WaitForSingleObject(thread->threadHandle, INFINITY, TRUE);
#endif
}
// This function will make the current thread attempt to give back the
// remaining time slice to the operating system
static void yield() {
#ifdef _POSIX
sched_yield();
#elif defined(_WIN32)
SwitchToThread();
#endif
}
// this function will start the thread
void start() {
#ifdef _POSIX
pthread_create(&pthreadObject, NULL, threadStart, NULL);
#elif defined(_WIN32)
threadHandle = CreateThread(NULL, 0, threadStart, thread, 0, &threadID);
#endif
}
// One must implement this function in order to use the thread class. When
// the start function of a thread is called, this function is executed.
virtual void run() = 0;
};

```

Figure 3.3: Non-Core Thread implementation

make fine-tuned adjustments to a specific platform implementation, they do not need to worry about having those changes accidentally erased or overwritten when they obtain source code for a new platform, nor do they have to worry about performing code merges when implementation updates become available.

At design level, the entire core solution for **Thread** is represented in Figure 3.8

```

#include <ThreadCore.h>

class Thread {
    friend class ThreadCore;
protected:
    ThreadCore * core;
public:
    inline Thread() { core = ThreadCore::createCore(); }
    inline virtual ~Thread() { ThreadCore::deleteCore(core); }

    inline static void waitFor(Thread * thread) { ThreadCore::waitFor(thread); }
    inline static void yield() { ThreadCore::yield(); }
    inline void start() { core->start(this); }
    virtual void run() = 0;
};

```

Figure 3.4: Thread implementation

```

// Forward declare the thread class.
#include <Thread.h>
class Thread;

// Abstract core class for Thread
class ThreadCore {
public:
    // Provide a platform independent way for a thread core to be created
    static ThreadCore * createCore();
    // A safe-delete for thread cores. More functionality should be added.
    inline static void deleteCore(ThreadCore * core) {
        if (core) delete core;
    }
    // Constructor and destructor, don't need to do anything
    inline ThreadCore() { }
    inline virtual ~ThreadCore() { }

    // The following functions are all platform dependent in their operations,
    // so make any platform-specific instance implement these functions.
    static void yield();
    static void waitFor(Thread * thread);
    virtual void start(Thread * thread)=0;
};

```

Figure 3.5: ThreadCore implementation

using the UML notation. From this figure, it can be noticed that this is a particular application of the generic core design shown in Figure 3.1.

```

#include <ThreadCore.h>
// Wrapper class for a operating-system thread.

// Check to make sure that the user wants to compile for win32 systems
#ifdef USE_WIN32THREADCORE

#include <ThreadCore.h>
#include <windows.h>
class Win32ThreadCore : ThreadCore {
protected:
    DWORD threadID;        // thread id
    HANDLE threadHandle;   // thread handle
    static DWORD threadStart(void * win32Param) {
        ((Thread*)win32Param)->run();
        return 0;
    }
public:
    inline Win32ThreadCore() { }
    inline virtual ~Win32ThreadCore() { }

    virtual void start(Thread * thread) {
        threadHandle = CreateThread(NULL, 0, threadStart, thread, 0, &threadID);
    }
};
// Simple implementations
ThreadCore * ThreadCore::createCore() {
    return new Win32ThreadCore;
}
void ThreadCore::waitFor(Thread * thread) {
    Win32ThreadCore * core = (Win32ThreadCore*)thread->core;
    WaitForSingleObjectEx(core->threadHandle, INFINITY, TRUE);
}
void ThreadCore::yield() {
    SwitchToThread();
}
#endif

```

Figure 3.6: Win32ThreadCore implementation

3.4 Router Example

The intuitive use of routers can be grasped from a **File** class. Practically every operating system uses files and allows files to be adequately represented using a standard string. However, the common operations one might expect from a **File** class are not implemented using the same functions on every platform. For example, to determine all the files in a directory on a POSIX file system, one would use the **opendir**, **readdir**, and **closedir** function calls. But on a Win32 file system, one would use the **FindFirstFile**, **FindNextFile**, and **FindClose** function calls.

A useful **File** class would contain several operations, but for the sake of brevity, we limit the scope of this example to a constructor, a destructor, an **exists** function (which determines if the given file exists in the file system), and an **isDirectory**

```

// Make sure the user wants to compile for a POSIX compliant system
#ifdef USE_POSIXTHREADCORE

#include <ThreadCore.h>
#include <pthread.h>
#include <sched.h>

// Posix implementation of the thread core class
class PosixThreadCore : ThreadCore {
protected:
    pthread_t posixThread;
    static void * threadStart(void * pthreadParam) {
        Thread * thread = (Thread*)pthreadParam;
        thread->run();
        return 0;
    }
public:
    // empty constructor and destructor
    inline PosixThreadCore() { }
    inline virtual ~PosixThreadCore() { }

    // start the thread
    virtual void start(Thread * thread) {
        pthread_create(&posixThread, NULL, threadStart, thread);
    }
};

// Simple implementations, but since we do not want a lot of preprocessor
// statements everywhere, we implement them in this file. Implementing these
// functions in this file also helps to ensure that no more than one
// threadcore implementation is linked

ThreadCore * ThreadCore::createCore() {
    return new PosixThreadCore;
}

void ThreadCore::waitFor(Thread * thread) {
    PosixThreadCore * core = ((PosixThreadCore*)thread)->core;
    pthread_join(core->posixThread, NULL);
}

void ThreadCore::yield() {
    sched_yield();
}

#endif

```

Figure 3.7: PosixThreadCore implementation

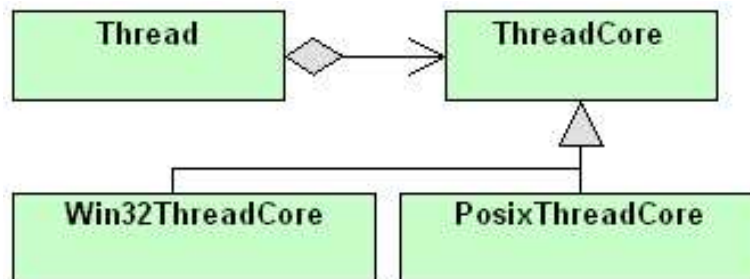


Figure 3.8: UML Diagram for Thread core

function (which determines if the given file represents a directory in the file system). Figure 3.9 shows how a File class can be written using the preprocessor to detect the correct build platform. Figure 3.10 presents the implementation of a **File** class which uses the router technique discussed in Section 3. As should be noticed, all the operations are passed directly to the **FileRouter** class.

The **FileRouter** class is shown in Figure 3.11. As was explained in Section 3, router classes represent an optimization over core classes as they do not require virtual functions. Router classes also do not require any platform-dependent data members.

```

#include <string>
#ifdef _POSIX
    #include <unistd.h>
    #include <dirent.h>
    #include <sys/stat.h>
#elif defined(_WIN32)
    #include <windows.h>
    #include <sys/stat.h>
#else
    #error Couldn't detect correct OS
#endif
using namespace std;
// File class, similar to java.io.File, though lacking functionality
class File {
protected:
    string path; // the path to the file
public:
    inline File(const string & filePath) : path(filePath) { }
    inline virtual ~File() { }
    // ensure that a File is a directory
    inline bool isDirectory() const {
#ifdef _POSIX // if on posix system
        struct stat sbuf;
        if (stat(path.c_str(), &sbuf) == -1) return false;
        return (sbuf.st_mode & _S_IFDIR) != 0;
#elif defined(_WIN32)
        struct _stati64 sbuf;
        if (_stati64(path.c_str(), &sbuf) == -1) return false;
        return (sbuf.st_mode & _S_IFDIR) != 0;
#endif
    }
    // make sure that a file exists
    inline bool exists() const {
#ifdef _POSIX // if on posix
        struct stat sbuf;
        return stat(path.c_str(), &sbuf) != -1;
#elif defined(_WIN32)
        return GetFileAttributes(path.c_str()) != INVALID_FILE_ATTRIBUTES;
#endif
    }
};

```

Figure 3.9: Non-System File implementation

The **PosixFileRouter** class implementation is shown in Figure 3.12. Just like in the Core example, one could implement a file router using Win32 concurrently with

the POSIX file router.

Just as with core classes, it may appear that less work is necessary if one used the method shown in Figure 3.9 than when using a router, but the same consequences indicated in Section 3 in relation with cores could be observed for routers when a larger perspective is considered.

From a design level perspective, the entire router solution for **File** is shown in UML notation in Figure 3.13. From this figure it can be noticed that this is a particular application of the generic router design shown in Figure 3.2.

```
#include <string>
#include <FileRouter.h>

// Platform-independent wrapper for the FileRouter class
class File {
protected:
    std::string path;
public:
    // Simply pass off all operations to the FileRouter class
    inline File(const std::string & filePath) : path(filePath) { }
    inline ~File() { }
    inline bool exists() const { return FileRouter::exists(path); }
    inline bool isDirectory() const { return FileRouter::isDirectory(path); }
};
```

Figure 3.10: File implementation

```
#include <string>

// Class to handle all File operations in a platform-dependent manner
class FileRouter {
public:
    static bool exists(const std::string & path);
    static bool isDirectory(const std::string & path);
};
```

Figure 3.11: FileSystem implementation

```

// Check to make sure that the user wants a Posix compatible implementation
#ifdef USE_POSIXFILEROUTER

#include <FileRouter.h>
#include <sys/stat.h>

// check to see if a file (path) is a directory
bool FileRouter::isDirectory(const std::string & path) {
    struct _stat sbuf;
    if (stat(path.c_str(), &sbuf) == -1) return false;
    return (sbuf.st_mode & S_IFDIR) != 0;
}

// check to see if a file (path) exists
bool FileRouter::exists(const std::string & path) {
    struct stat sbuf;
    return stat(path.c_str(), &sbuf)!=-1;
}

#endif

```

Figure 3.12: Posix File System implementation

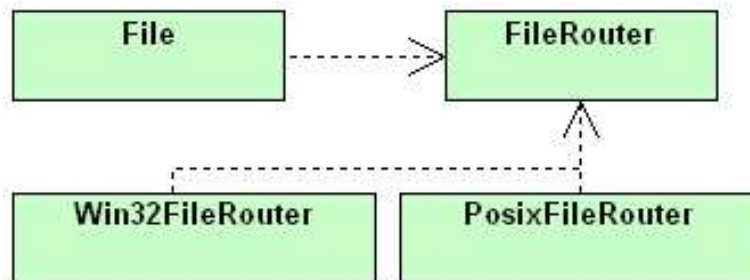


Figure 3.13: UML Diagram for File router

Chapter 4

The JTK and PCM-Client/Server

With the idea for cores and routers solidified, a functional example was necessary to truly expose the potential of these methods. As such, a library known as the ‘J’ Toolkit (**JTK**) was developed using cores and routers. An application was also developed that used the **JTK**. Section 4.1 gives details on the **JTK** and Section 4.2 gives details on the sample application made with the **JTK**.

4.1 JTK

The **JTK** needed to address several issues commonly dealt with in cross-platform library development, as well as less common issues, as follows:

- Use cores and routers
- Provide the same ease of use as any other cross-platform library
- Allow application code to be write-once, compile-anywhere
- Give true low-level access to a system
- Allow for easy expansion (adding new platform support), extension (adding new functionality), and optimization

Using cores and routers is an obvious requirement. Providing ease of use is important because it shows that the use of cores and routers does not make application development any harder than with other cross-platform development methods. For

the **JTK**, it was decided to model many of the classes after their corresponding implementation in Java. Making application code “write-once, compile-anywhere” is a common goal for virtually all cross-platform APIs. Some cross-platform libraries do not give access to the full potential of the underlying operating/embedded system, some say this is due to the design of the library. The JTK needed to show that a cross-platform library written using cores and routers could deliver all the features one would have if they were using platform-dependent code instead of a platform-independent API. The ability to expand, extend and optimize are also important because these are three critical areas examined with any cross-platform design technique and library.

Keeping in mind the ideas mentioned above, the **JTK** provides several packages of classes (implemented as namespaces in C++). Figure 4.1 gives a diagram of all the packages of the **JTK**. The **jtk** package is presented in Section 4.1.1, the **jtk::io** package is described in Section 4.1.2, the **jtk::media** package is explained in Section 4.1.3, the **jtk::net** package is presented in Section 4.1.4, the **jtk::os** package is described in Section 4.1.5, and the **jtk::util** package is explained in Section 4.1.6.

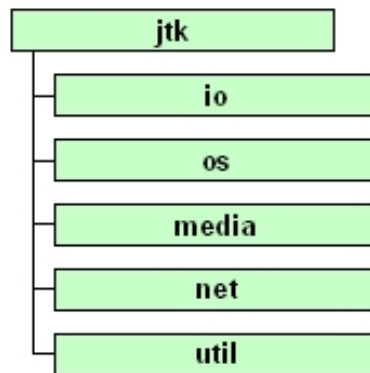


Figure 4.1: The JTK Package hierarchy

4.1.1 Package: **jtk**

The **jtk** package contains the base class for all others classes in the library, **jtk::Object**, as well as a base class for all run-time exceptions, **jtk::Exception**. Having a common

base class for runtime exceptions provides an elegant method to deal with exceptions using C++. Wrapper classes for C++ primitives reside in the **jtk** package. In combination with all these classes, all sub-packages of the **JTK** are nested in the **jtk** package. Figure 4.2 shows all the classes of the **jtk** package.

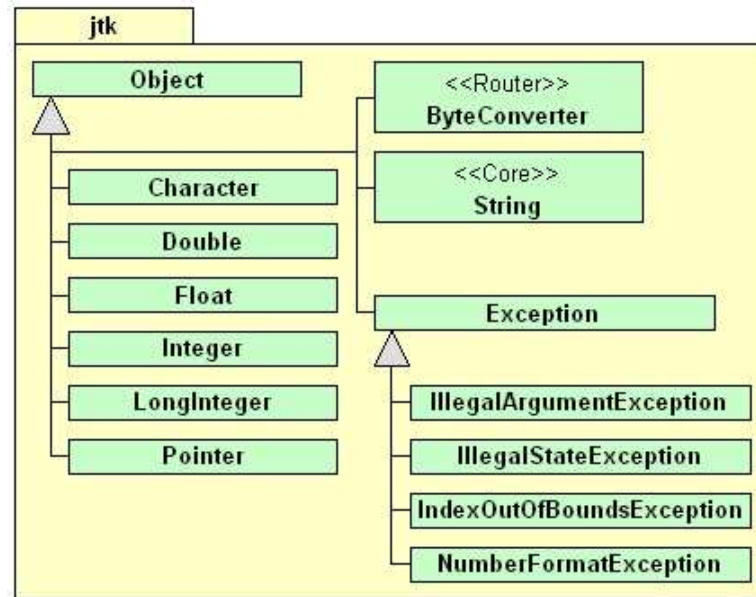


Figure 4.2: The **jtk** package

4.1.2 Package: **jtk::io**

The **jtk::io** package contains all necessary base classes to perform low-level input and output (I/O). Included are classes to represent streams of information (e.g. **InputStream** and **OutputStream**), a core-based **FileDescriptor** class, a router-based **File** class, and a **Pipe** class. Figure 4.3 shows the class hierarchy for the **jtk::io** package.

The **FileDescriptor** class serves as an abstraction for file descriptors on different platforms. On a windows system, a file descriptor is represented as a **HANDLE**, while on a UNIX/Linux system, a file descriptor is simply an **int**.

The **File** class is a full implementation of the router example shown in Chapter 3. It provides almost all the functionality one would expect from a file on any operating

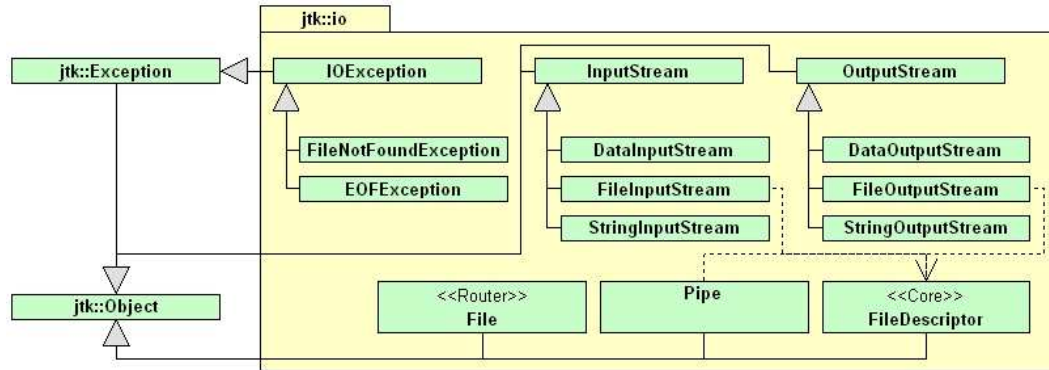


Figure 4.3: The `jtk::io` package

system. And since a standard `string` can be used to represent a file to an operating system, the router solution can be used instead of the core solution.

4.1.3 Package: `jtk::media`

The `jtk::media` package provides low-level access to any media equipment on the computer. Currently, only writing to audio devices is supported, but that is more than what most general-use cross-platform APIs provide. This package gives possibly the best example of how powerful and elegant the **JTK** is because it was implemented using cores and routers. The `jtk::media` package provides a class to read Pulse Code Modulation (PCM) files, as well as a core-based `AudioDevice` class that can be used to play PCM files. The `jtk::media` package also provides exception classes for media devices, as well as an event-listener combination for audio device events. A full class diagram for the `jtk::media` package is shown in Figure 4.4.

Media APIs are commonly the most non-standard API on operating systems. For example, the Windows operating system allows the use of one of two custom built libraries, the Windows MultiMedia (WinMM) library or the Windows DirectSound library. On Linux, the operating system allows the use of either the Open Sound System (OSS) or the Advanced Linux Sound Architecture (ALSA) library. On Mac OS X, the operating system uses the custom built library Core Audio. This varies from other cross-platform APIs such as sockets due to how the complete difference the

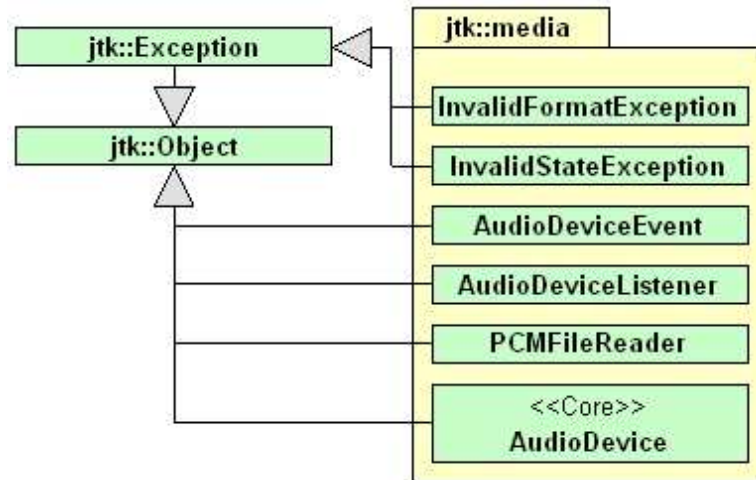


Figure 4.4: The `jtk::media` package

underlying operating system APIs for media access are implemented. Like other cross-platform libraries, the **JTK** hides these differences from the application programmer. However, it is in the ease of extension that shows how elegant the design of a library using cores and routers can be.

The **AudioDevice** is used to wrap the low-level access to sound cards. Very little functionality is implemented in the abstract core class due to the vast differences in sound API on operating systems.

4.1.4 Package: `jtk::net`

The `jtk::net` package gives programmer all the classes and functionality necessary to implement an application that uses TCP/IP or UDP networking. Classes are present to represent TCP sockets, TCP server sockets, and UDP sockets (all of which use cores). Since TCP sockets are stream based, a core-based input stream and a core-based output stream class are also provided. UDP sockets have a platform-indepent **DatagramPacket** class used to send and receive datagrams. The `jtk::net` package also provides address support using the **InternetAddress** and **IPv4Address** classes. The **IPv4Address** class uses a router-solution, since exactly four bytes are sufficient to represent an address using IPv4. A full class diagram for the `jtk::net` package is

shown in Figure 4.5.

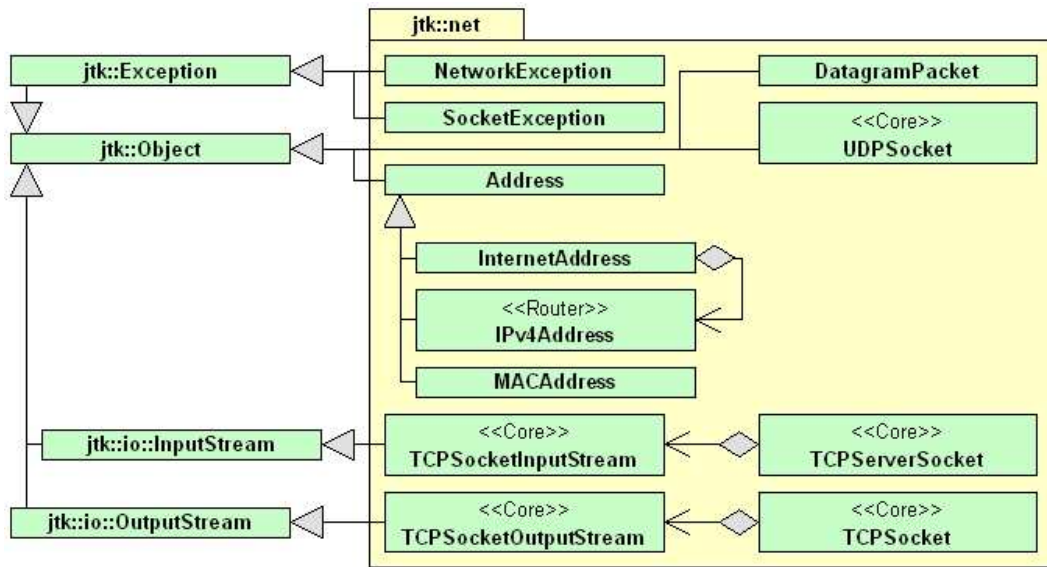


Figure 4.5: The `jtk::net` package

4.1.5 Package: `jtk::os`

Classes dealing primarily with the low-level functionality of an operating system are provided in the `jtk::os` package. One example is the core-based **Thread** class that represents a thread of execution on an operating system and allows an application developer to run multiple threads in their code. Core-based **Semaphore** and **Process** classes are also in this package, as well as a router-based **OperatingSystem** class which serves as the intermediary between the application and the environment of the **OperatingSystem**. A full class diagram for the `jtk::os` package is shown in Figure 4.6.

4.1.6 Package: `jtk::util`

While the `jtk::media` package is the best example for providing low-level access to an application programmer, the `jtk::util` package is the best example of giving the ability to easily **optimize** to the application programmer. The `jtk::util` package has

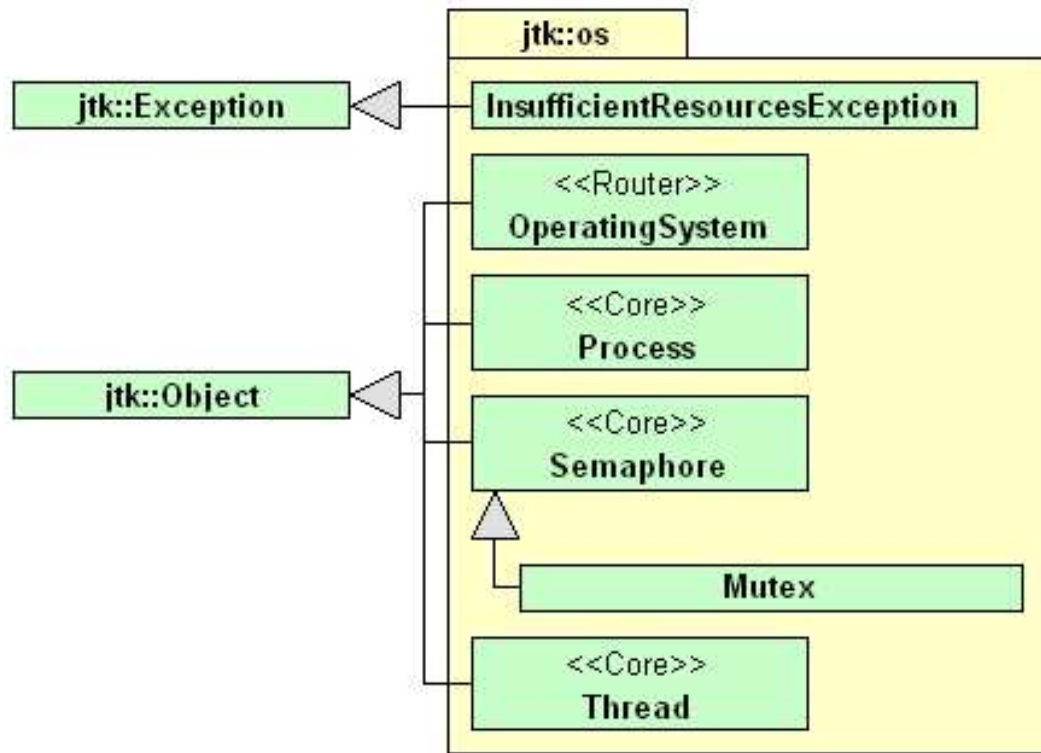


Figure 4.6: The `jtk::os` package

core-based container classes such as lists and vectors, as well as general use classes like the regular-expression `Pattern` and `Matcher` classes, or the core-based `Timer` class. A diagram of the `jtk::util` package is shown below in Figure 4.7.

The container classes default to using the standard template library (STL) classes for their implementation, but one could choose to make their own optimized `List` or `Map` class, and compile their own code in the place of the default STL implementation. This shows the power of optimization and customization using cores, a user can create their own customized/optimized implementation of a class, and simply drop it in, without worrying about damaging the default implementation provided.

The `Timer` class is core-based because high-precision timing is not handled the same way on every operating system. If the `Timer` is to be used on a Windows operating system, it must interface with the `PerformanceCounter` or multi-media timer, whereas on UNIX/Linux, the `Timer` class can simply use the `gettimeofday`

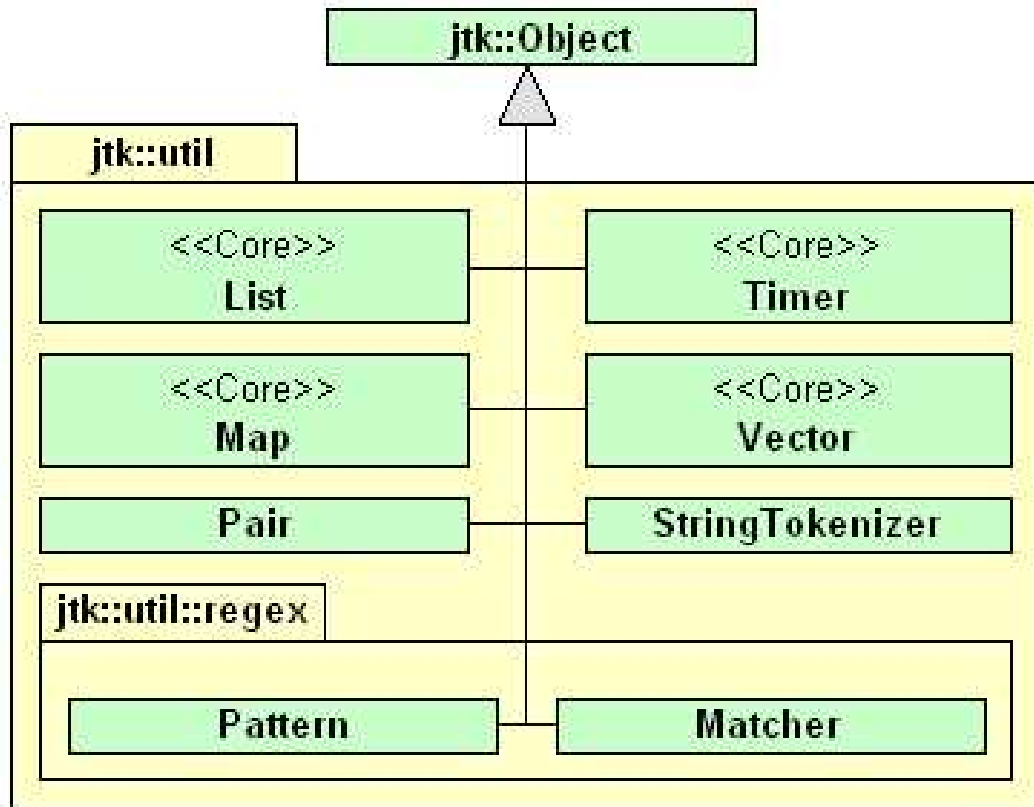


Figure 4.7: The `jtk::util` package

function. The **Timer** class is also an excellent example of how the **JTK** does not have to use the lowest-common-denominator of functionality between operating systems. On Windows, the granularity of timing accuracy is at best one millisecond. On most UNIX/Linux systems, an accuracy in tens/hundreds of microseconds can be achieved using the `gettimeofday` function.

4.2 PCM-Client/Server

In order to demonstrate the capabilities of the **JTK**, two sample applications were built. One is a *network-server* that acts as a limited file server and can stream Pulse Code Modulation (PCM) data to clients. The other application is the *network-client* that interfaces with the network-serve to browse remote directories and stream Pulse Code Modulation data.

4.2.1 PCM-Server

The server application was designed with the goal of maximizing use of platform-dependent concepts through the **JTK**. As such, the server was implemented using multiple threads, several semaphores, and both TCP and UDP networking. The server then relies heavily on the `jtk::net` and `jtk::os` packages. A flowchart for the behavior of the server is shown in Figure 4.8. It should be noted that more error checking is present in the code than there is shown in Figure 4.8. Error checking was removed from the flow diagram for brevity.

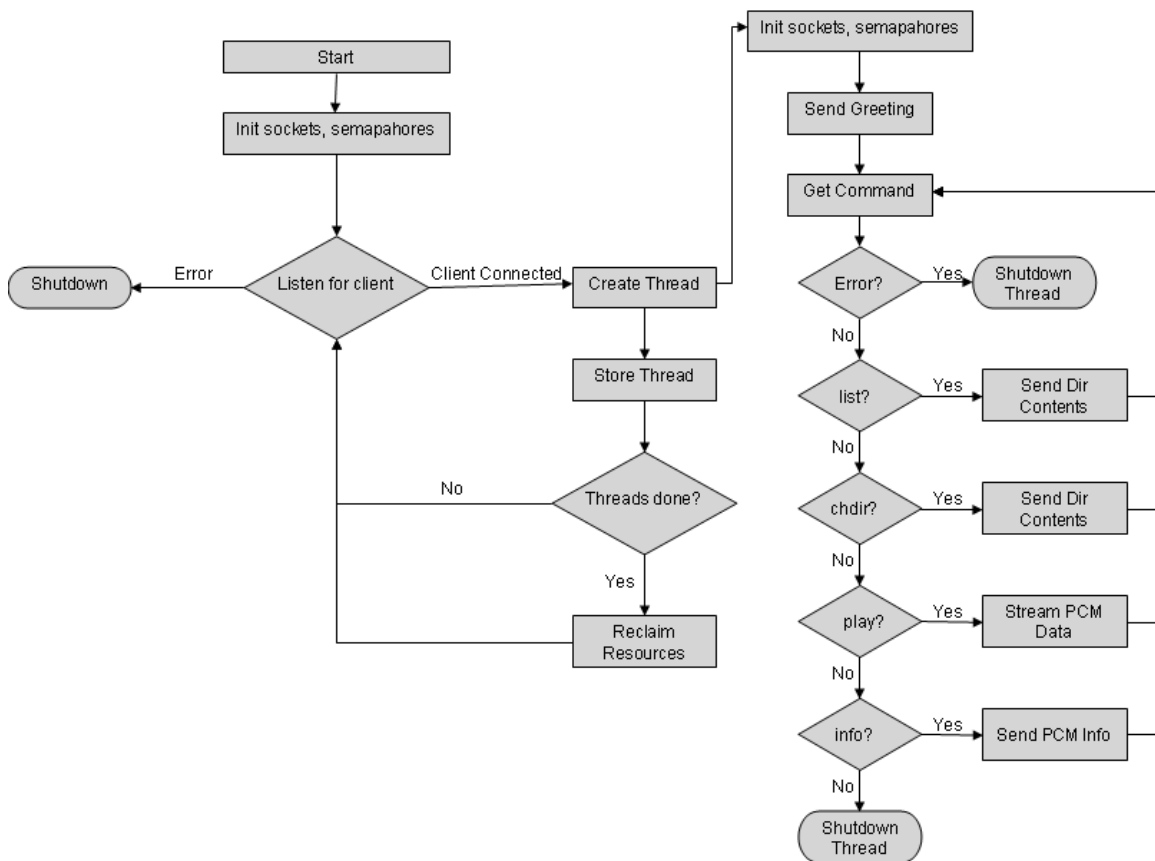


Figure 4.8: PCM-Server Flow Diagram

The server application is approximately three-hundred and fifty lines of code using the **JTK**. This amount is near optimum, even if the server used another cross-platform API in place of the **JTK**. If the server were to be written using platform-

dependent code in place of the **JTK**, the code size would expand to more than one-thousand lines. The **JTK**, with cores and routers, not only helps the development of cross-platform code, but can also make application-code as (if not more) concise and readable than any other cross-platform library. A full listing of the server code can be found at http://www.cse.unr.edu/~stuart/projects/thesis_app/server.tar.gz.

4.2.2 PCM-Client

A flow-chart of the PCM client is shown in Figure 4.9, and a discussion of the client follows.

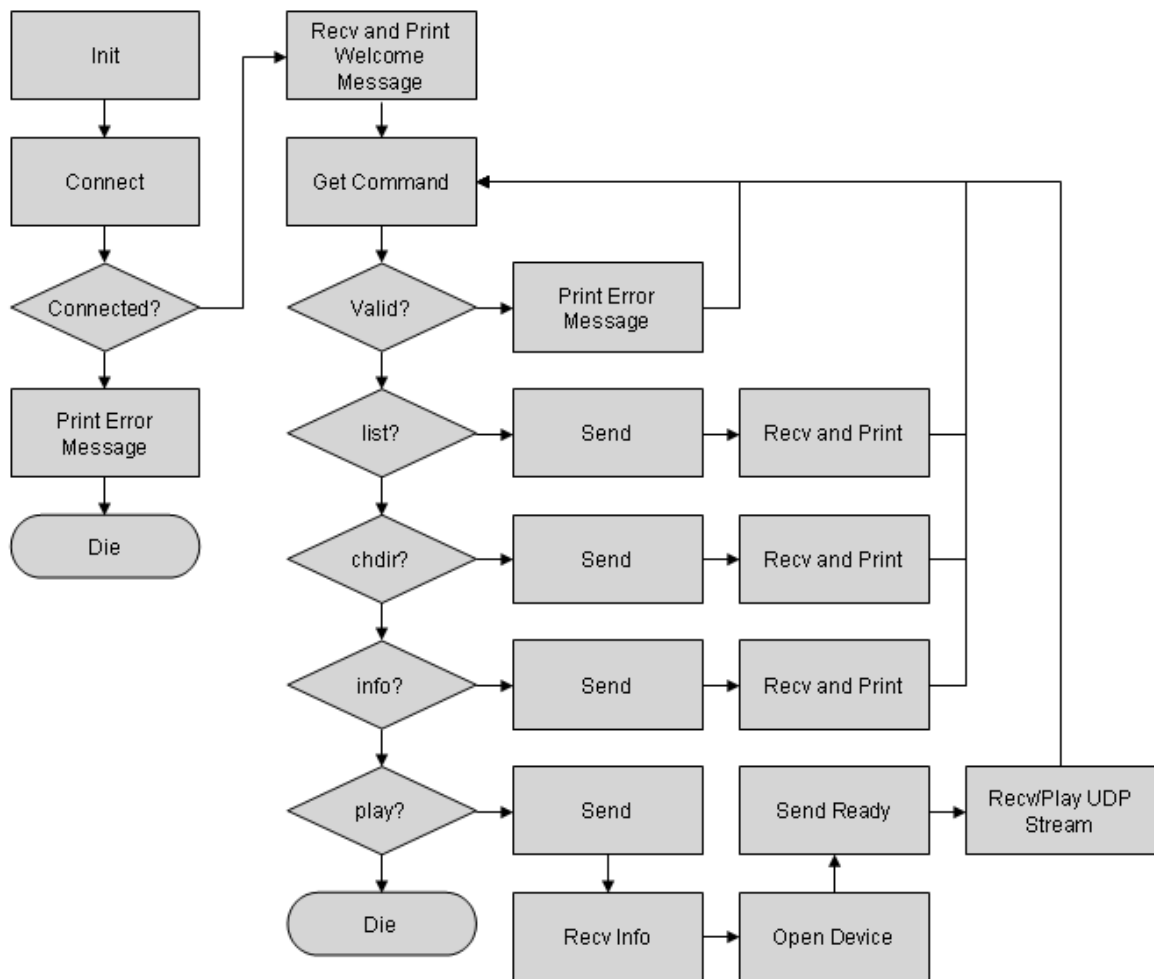


Figure 4.9: PCM-Client Flow Diagram

The client application was designed primarily as an interface to the server, and

also to show off the audio device capabilities of the the **JTK**. The audio-device capabilities were an important feature to test, being that those capabilities required the lowest latency and highest-performance of anything in the **JTK**. The client runs in a single thread, and uses standard TCP to talk to the server and UDP to stream PCM data from the server.

Since the client does not use multiple threads, it relies less on the `jtk::os` package than the server, but it still makes heavy use of the `jtk::net` package, as well as using the `jtk::net::AudioDevice` class.

The client code is approximately two-hundred and sixty lines of code (using the **JTK**). If the application was to be written using platform-dependent code in place of the **JTK**, the source code would swell to approximately eight- hundred lines. The complete source code for the client can be found at http://www.cse.unr.edu/~stuart/projects/thesis_app/client.tar.gz.

Chapter 5

Future Work

Several possibilities exist for future work on this topic. First and foremost, it is hoped that an elegant way can be found to rid the code of pointer lookups, which can be computationally expensive in some cases. Using separate development branches or making heavy use of the preprocessor can lead to better performance and, in some cases, this performance is crucial. From a compilers' and optimizations' standpoint, so far inheritance has been dealt with rather inefficiently. Inheritance adds sometimes unnecessary overhead, as pointer lookups can be expensive. However, when only a single subclass of an abstract class is implemented (as is the typical case with cores), it seems a compiler should optimize the code by “merging” the inherited class into the abstract class, therefore eliminating the overhead of virtual functions and such [11, 20]. Currently, no compiler exists (that we know of) that can do this, so writing this optimization is one potential direction of future work.

Another avenue of future work is the extension of the *JTK*. Currently, only Windows XP and Linux/UNIX are supported (partial support for Mac OS X is included). The goal of this thesis is to make software development on multiple branches elegant and efficient. Since the *JTK* provides a basic framework from which to start, it is hoped that others will choose to add on to it and support other platforms such as Mac OS X, and miscellaneous embedded systems.

Another interesting direction of future work is in the construction of a programming environment conducive to the development of cross-platform code. Research has been done at the University of Nevada, Reno, on a concept known as stratified

programming [16] where, in essence, code is organized into strata and substrata, and the developer, based on his or her objectives in a given context, can choose to hide all strata beneath a certain threshold. This concept can be expanded to cross-platform source code development, where instead of only using strata, one also chooses to display which target-platform's code one wants to see.

Chapter 6

Conclusions

Object oriented programming (C++, C#, Java) has become popular, but old habits from the days of imperative programming (C) still have a strong influence on implementation styles. Several libraries were studied that have similarities with the method we proposed. In particular, [23] and [6] use inheritance for platform specific code, but in contrast to our approach they tend to:

- Use multiple levels of inheritance, when one generally suffices;
- Provide only one implementation of the inherited class, and specifically reference it in various places;
- Separate code development and releases into platform-specific branches; and
- Use a **void** pointer (which becomes a pointer to a structure) for member variables instead of having the variables stored in the inherited class.

The library written (**JTK**) shows that the ideas of cores and routers are both easy to understand and easy to implement. Expansion and extension of code that uses cores and routers has also been proven to be an easy and straight-forward endeavor.

The concepts of cores and routers presented in this thesis and the software development method they promote are aimed at creating higher-quality cross-platform code. Simplicity and efficiency are desirable qualities in programming which can be achieved with the development solutions proposed. While improved code readability, code sharing, and program structure can certainly benefit from these solutions, future

work should focus on code optimization and larger-scale application of the proposed methods.

Bibliography

- [1] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. A Pattern Language, 1997.
- [2] L.C. Banerjee, T. DeFanti, and S. Mehrotra. Realistic Cross-Platform Haptic Applications Using Freely Available Libraries. *Proceedings of the 12th IEEE International Symposium on Haptic Interfaces for Virtual Environments and Teleoperator Systems*, pages 282–289, 2004.
- [3] BOOST. Chapter 10. Boost.Threads. <http://www.boost.org/doc/html/threads.html>, Accessed June 6. 2005.
- [4] T.C. Brooke. Development of a Distributed, Cross-Platform Simulator. *Proceedings of ACM SIGADA*, pages 12–21, 2002.
- [5] Eldos Corporation. MsgConnect: There is a world to connect. <http://msgconnect.com>, Accessed March 2005.
- [6] E. Crahen. Netinformations Computer Guide: “ZThreads”. <http://zthread.sourceforge.net>, Accessed March 2005.
- [7] M.A. Cusumano and D.B. Yoffie. What Netscape Learned from Cross-Platform Software Development. *Communications of the ACM*, 42:72–78, 1999.
- [8] S.M. Dascalu, A. Pasculescu, J. Woolever, E. Fritzinger, , and V. Sharan. Stratified Programming Integrated Development Environment (SPIDER). *Proceedings of the 12th International Conference on Intelligent and Adaptive Systems and Software Engineering*, pages 227–232, 2003.
- [9] DOFactory. Design Patterns. <http://www.dofactory.com/Patterns/Patterns.aspx>, Accessed June 7. 2005.
- [10] M. Franz. Dynamic Linking of Software Components. *IEEE Computer*, 30(3):74–81, 1997.
- [11] A. Gal, S. Wolfgang, and O. Spinczyk. On Minimal Overhead Operating Systems and Aspect-Oriented Programming. *Proceedings of the 4th ECOOP Workshop on Object-Oriented*, 2001.
- [12] GNU. GNU Make. <http://www.gnu.org/software/make/make.html>, Accessed June 6. 2005.

- [13] E. Goman, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.
- [14] GTK+. The GIMP Toolkit. <http://www.gtk.org>, Accessed Jan. 28 2005.
- [15] I.J. Hayes. Towards Platform-Independent Real-Time Systems. *Proceedings of the 2004 Australian Software Engineering Conference*, pages 1–9, 2004.
- [16] J. Jusayan. SPINDLE: The Stratified Programming INtegrated DeveLopment Environment. 2004.
- [17] S. Li, J. Xu, and L. Deng. Periodic Partial Validation: Cost-Effective Source Code Validation Process in Cross-Platform Software Development Environment. *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing*, pages 401–406, 2004.
- [18] H. Nishimura, C. Timossi, and J.L. McDonald. Cross-Platform SCA Component Using C++ Builder and Kylix. *Proceedings of the Particle Accelerator Conference*, pages 2385–2386, 2003.
- [19] OpenSG. OpenSG Home. <http://www.opensg.org>, Accessed March 2005.
- [20] U. P. Shultz. Partial Evaluation for Class-Based Object-Oriented Languages. *Proceedings of the 2nd Symposium on Programs as Data Objects*, pages 173–197, 2000.
- [21] Hawk Software. HawkNL (Hawk Network Library). <http://www.hawksoft.com/hawknl>, Accessed March 2005.
- [22] POSIX Threads. POSIX Threads Links. <http://www.humanfactor.com>, Accessed April 2005.
- [23] Trolltech. QT Application Framework, Cross-Platform C++ GUI Development and Embedded Systems Solutions. <http://www.trolltech.com>, Accessed Jan. 28 2005.
- [24] UML. OMG’s UML resource page. <http://www.omg.org/uml>, Accessed March 2005.
- [25] UNIX. UNIX History. <http://www.levenez.com/unix>, Accessed June 6. 2005.
- [26] D. Waugh and W.J. Phillips. Cross-Platform Help Products: The Andyne Solution. *Proceedings of the IEEE Professional Communication Conference*, pages 86–88, 1995.
- [27] B. Westphal, F. Harris, and S. Dascalu. Snippets: Support for Drag-and-Drop Programming in the Redwood Environment. *Journal of Universal Computer Science*, 10(7):859–871, 2004.