Towards A Unified Approach for Cross-Platform Software Development

Jeffery A. Stuart, Sergiu M. Dascalu, Frederick C. Harris Jr.

Department of Computer Science and Engineering
University of Nevada, Reno - Reno, Nevada, 89557 USA
{stuart, dascalus, fredh}@cse.unr.edu

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 paper 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 in the paper.*

Keywords: Cross-platform development, cores, routers

1 Introduction

Cross-platform software development is an intricate and demanding activity [1], [2], [3], [4], [5], [6], [7]. 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 [8]. 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. The authors have experienced the above when working on various software projects [9], [10], [11].

Starting from these observations, this paper 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 C++. 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 in C++ is suggested. Although the idea of the proposed approach might seem obvious at the first sight, as far as we know 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 [12], [13], and [14]. The proposed core and router solutions are illustrated in the paper through several examples. Details about the intended meaning of the specific terms used (*cores* and *routers*) are also provided.

The remainder of this paper is organized as follows: Section 2 presents background information on current cross-platform software development practices, design patterns, and related issues, Section 3 introduces the concepts of cores and routers intended to help solve these issues, Section 4 gives an example of a core-based development solution, Section 5 presents an example of a router-based solution, Section 6 outlines several directions of future work, and Section 7 concludes the paper with a summary of the paper's contributions.

2 Cross-Platform Software Development: Practices and Related Issues

Currently, there are two common practices (approaches, methods, or styles) used for cross-

platform API development. The first approach involves the use of separate segments (branches) of code, each written for a specific target. For example, [15] supports four operating systems (Windows, Vanilla Linux, Irix, and Irix 64), with a total of five compilers (Visual Studio 6 with Intel Compiler, Visual Studio 7, g++ 3.2 and above, SGI CC, and SGI 64 bit CC). The second approach is characterized by extensive use of preprocessor commands, which leads to several shortcomings, as outlined below.

In the first approach, although the packaged code is generally readable and the code well-structured, the possibility of sharing code across platforms is minimal, which is not a desirable feature. Furthermore, the numerous (and usually large) branches of code are hard to manage from a development point of view. Examples of API packages representative for this approach are [12], [13] and [14].

In the second approach, the 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 [16], [17], and [18]. As a matter of fact, the POSIX Threads package employs both methods, but we consider it a stronger fit for the "separate branches" category [14].

Starting from the above observations and driven by practical software development needs, we looked into the possibility of writing cross-platform code in a way that addresses 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 question we tried to answer is "how could the readability of the code be improved and the code sharing maximized while keeping the software organized?" The solution we suggest is based on two simple, yet efficient development solutions, referred to as cores and routers. These are in fact two design constructs aimed at C++ implementation. From the experience gained in writing code for a general-purpose C++ API and from using it in actual software development projects [9] we argue 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 C++ code. The concepts of cores and routers are introduced next while examples of their use are given later in the paper.

3 Cores and Routers

A core is a generic code development solution that can be represented using the UML notation [19] as shown in Figure 1. In this figure a class (ClassA) intended for cross-platform development relies on the services of the abstract class ClassACore which, in turn, has specialized platform-dependent implementations in the subclasses Platform1ClassACore, Platform2-ClassACore, 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, we decided to use this name to highlight their pragmatic significance.

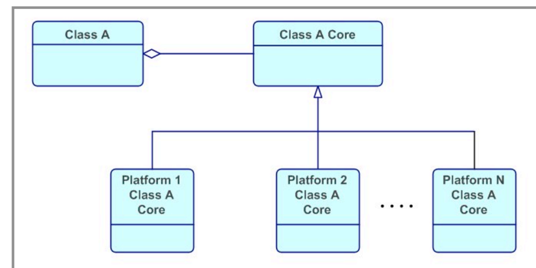

Fig. 1 UML diagram of the generic core solution

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 4 of the paper.

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 2 using the UML notation. In Figure 2, ClassB intended for cross-platform software development relies on the services of the concrete class ClassBRouter which, in turn, is associated with the platform-dependent implementation classes Platform1ClassBRouter, Platform2ClassBRouter, 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.

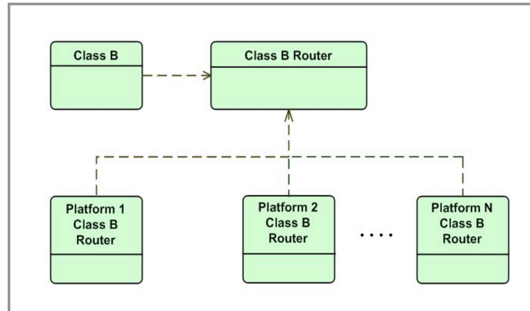

Fig. 2 UML diagram of the generic router solution

Note also that, in practical terms, our proposed generic router solution consists of a wrapper (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).

Cores and routers borrow concepts from the Bridge and Factory design patterns [22]. 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. Nevertheless, both cores and routers are new design solutions on their own.

Technically speaking, the main router class could be eliminated from the design presented in Figure 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 the paper (Section 5), actual base router classes may not always be necessary.

An example of using the router solution is provided in Section 4 of the paper. 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.

4 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 `pthread_t`, Win32 threads use `HANDLE`, Solaris Unix threads use `thread_t`, and so forth.

The expected functions for a Thread class are outlined as follows. First, a constructor and 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 is a previous implementation of the `Thread` class using the preprocessor 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 a core comes in. As explained in Section 3, a wrapper class, an abstract core class, and implemented core classes are needed. The implementation for the `Thread` wrapper class is shown in Figure 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 5. The platform-independent functionality has been implemented but, as it 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 6 is the concrete class `Win32ThreadCore`, which implements all the remaining (platform-specific) code necessary to use a thread. Figure 7 shows the concrete class `PosixThreadCore`. Similar to the class `Win32ThreadCore`, 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 2, one must consider things 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 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 8 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 1.

5 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.

```
#ifdef _POSIX
  #include <pthread.h>
  #include <sched.h>
#elif defined(_WIN32)
  #include <windows.h>
#else
  #error Couldn't detect correct OS
#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:
    inline Thread()           { }
    inline virtual ~Thread()  { }
    // even though the code is the same, the
    // parameters aren't, so we must have
    // different implementations of this
    // function
    #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
    inline 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
    inline static void yield() {
      #ifdef _POSIX
        sched_yield();
      #elif defined(_WIN32)
        SwitchToThread();
      #endif
    }
    // this function will start the thread
    inline 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. Whenever
    // the "start" function of a thread is
    // called, this function is executed. The
    // thread will cease execution shortly
    // after this function finishes
    virtual void run() = 0;
};
```

Fig. 3 Non-core implementation of Thread

```
class Thread
{
  protected:
    ThreadCore * core;
  public:
    // Constructor and destructor
    inline Thread()
      { core = ThreadCore::createCore(); }
    inline virtual ~Thread()
      { ThreadCore::deleteCore(core); }

    // Allow access to our
    // "ThreadCore" if wanted
    inline ThreadCore * getCore()
      { return core; }
    // The following functions all rely on the
    // thread core to perform their tasks
    inline static void waitFor(Thread * thread)
      { ThreadCore::waitFor(thread); }
    inline static void yield()
      {ThreadCore::yield(); }
    inline void start()
      { core->start(this); }
    // One must implement this function in
    // order to use the thread class. Whenever
    // the "start" function of a thread is
    // called, this function is executed. The
    // thread will cease execution shortly
    // after this function finishes.
    virtual void run() = 0;
};
```

Fig. 4 Thread wrapper implementation in the core solution

```
// Forward declare the thread class. We don't
// actually do anything with it in this file so
// we don't need to #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 such as error-checking to ensure
  // that the core being deleted
  // is not alive or holding any
  // mutexes/semaphores/monitors etc.
  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;
};
```

Fig. 5 ThreadCore implementation

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

```
#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>
// Win32 impl. of the ThreadCore class
class Win32ThreadCore : ThreadCore
{
  protected:
    DWORD threadID; // Win32 thread id
    HANDLE threadHandle; // Win32 thread handle
    // Start the thread running. Error
    // checking should be added.
    static DWORD threadStart(void * win32Param)
    {
      ((Thread*)win32Param)->run();
      return 0;
    }
  public:
    // Empty constructor and destructor
    inline Win32ThreadCore() { }
    inline virtual ~Win32ThreadCore() { }

    // Function to start the thread running
    virtual void start(Thread * thread) {
      threadHandle =
        CreateThread(NULL, 0,
                     threadStart,
                     thread, 0, &threadID);
    }
};
// Simple implementations, but since we do
// not want a lot
// of preprocessor statements, we simply
// implement these
// functions in this file.
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
```

Fig. 6 Win32ThreadCore implementation

on a POSIX file system, one would use the `opendir`, `readdir`, and `closedir` function calls. But on 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` function which determines if the given file represents a directory in the file system. Figure 9 shows how a `File` class can be written using the preprocessor to detect the correct build platform. Figure 10 presents the implementation of a `File` class which uses the router technique discussed in Section 3.

```
// 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 ThreadCore 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
```

Fig. 7 PosixThreadCore implementation

As it can be noticed, all the operations are passed directly to the class `FileRouter`.

The declaration of the class `FileRouter` is shown in Figure 11. As was explained in Section 3, router classes represent an optimization over core classes as they do not require any virtual functions or pointer lookups. Router classes also do not require any platform-dependent member variables.

The `PosixFileRouter` class implementation is shown in Figure 12. Just like in the Core example, one could implement a file router using Win32 concurrently with the POSIX file router.

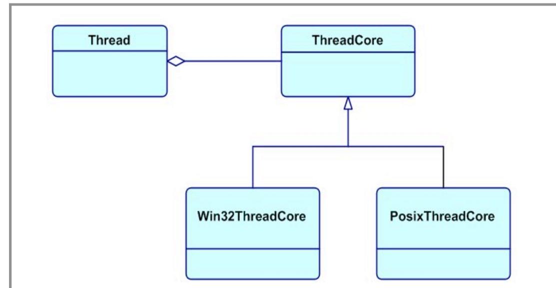

Fig. 8: UML diagram of cross-platform core solution for Thread

```
#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
// currently lacking any
// significant functionality
class File
{
  protected:
    string path; // the path to the file
  public:
    inline File(const string & filePath)
      : path(filePath) { }
    inline virtual ~File() { }
    // This function simply checks to ensure
    // that a "File" is actually a directory
    // (true would mean that the file exists
    // and is a directory
    inline bool isDirectory() const {
      #ifdef _POSIX // if posix, then use stat
      struct stat sbuf;
      if (stat(path.c_str(), &sbuf) == -1)
        return false;
      return (sbuf.st_mode & _S_IFDIR) != 0;
      #elif defined(_WIN32) // else use stati64
      struct _stati64 sbuf;
      if (_stati64(path.c_str(), &sbuf) == -1)
        return false;
      return (sbuf.st_mode & _S_IFDIR) != 0;
      #endif
    }
    // this function simply checks to make sure
    // that a file exists
    inline bool exists() const {
      #ifdef _POSIX // if on posix, use stat
      struct stat sbuf;
      return stat(path.c_str(), &sbuf) != -1;
      #elif defined(_WIN32) // else, use Win32
      return GetFileAttributes(path.c_str())
             != INVALID_FILE_ATTRIBUTES;
      #endif
    }
};
```

Fig. 9 Non-router implementation of File

Just as with core classes, it may appear to be less work to do when using the method shown in Figure 9 (than when using a router), but the same consequences indicated in Section 4 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 13. From this figure it can be noticed that this is a particular application of the generic router design shown in Figure 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); }
};
```

Fig. 10 File implementation in router solution

```
#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);
};
```

Fig. 11 FileRouter implementation

6 Future Work

Several possibilities exist for future work on this topic. First and foremost, we would like to find an elegant way to rid the code of pointer lookups. 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 compiler and optimization 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 know to optimize the code by "merging" the inherited class into the abstract class, therefore eliminating the overhead of virtual functions and such [20], [21]. Currently, no compiler exists (that we

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

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

// Notice that we don't need to define a
// PosixFileRouter class. For more complex
// operations, a new Router class might be
// desirable, but none of the functions we
// implement in this case really call for it.

// 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
```

Fig. 12 PosixFileRouter implementation

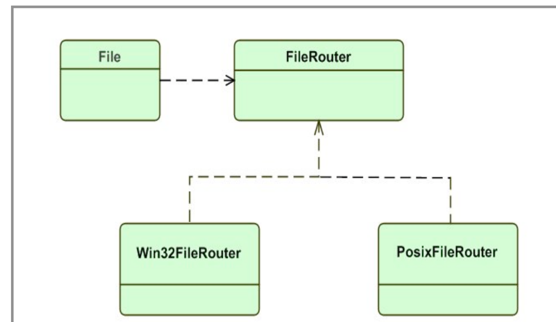

Fig. 13 UML diagram of cross-platform router solution for File

of) that can do this, so writing this optimization is one potential direction of future work. Another direction of further work, which we have already taken, is the development a general use C++ library that makes use of the solutions proposed and demonstrates their utility. Yet perhaps the most 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 UNR on a concept known as stratified programming [10] 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 could possibly be expanded to cross-platform source code development such that instead of hiding and showing strata, one would hide and show platform-specific code.

7 Conclusions

Object oriented programming (C++) has become popular, but old habits from the days of imperative programming (C) still have a strong influence on implementation styles. Several libraries that we have studied have similarities with the method we proposed. In particular, QT [12] and ZThreads [16] 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* (which becomes a pointer to a structure) for member variables instead of having the variables stored in the inherited class.

The concepts of cores and routers presented in this paper and the software development method they promote are aimed at creating higher-quality cross-platform code. Simplicity and efficiency are desirable qualities in programming that we believe can be achieved with the development solutions proposed. While improved code readability, code sharing, and program structure can certainly benefit from these solutions, our future work needs to focus on code optimization and larger-scale application of the proposed approach.

References

[1] Waugh, D. and Phillips, W.J. (1995) “Cross-Platform Help Products: the Andyne Solution”, Procs. of the IEEE Professional Communication Conference, 86-88.

[2] Franz, M. (1997) “Dynamic Linking of Software Components”, IEEE Computer, 30 (3), March 1997, 74-81.

[3] Brooke, T.C. (2002) “Development of a Distributed, Cross-Platform Simulator”, Proceedings of ACM SIGADA 2002, 12-21.

[4] Nishimura, H., Timossi, C., and McDonald, J.L. (2003) “Cross-Platform SCA Component Using C++ Builder and Kylix”, Procs. of the Particle Accelerator Conference (PAC 2003), 2385-2386.

[5] Banerjee, L.C., DeFanti, T., Mehrotra, S. (2004) “Realistic Cross-Platform Haptic Applications Using Freely Available Libraries”, Procs. of the 12th IEEE Intl. Symp. on Haptic Interfaces for Virtual Env. and Teleoperator Systems, 282-289.

[6] Hayes, I.J. (2004) “Towards Platform-Independent Real-Time Systems”, Proceedings of the 2004 Australian Software Engineering Conf. (ASWEC’04), 1-9.

[7] Li, S., Xu, J, and Deng, L. (2004) “Periodic Partial Validation: Cost-Effective Source Code Validation Process in Cross-Platform Software Development Environment”, Proceedings of the IEEE Pacific Rim Intl. Symposium on Dependable Computing, 401-406.

[8] Cusumano, M.A. and Yoffie, D.B (1999) “What Netscape Learned from Cross-Platform Software Development”, ACM Comm., 42 (10), 72-78.

[9] Jusayan, J. (2004). “SPINDLE: The Stratified Programming INtegrated DeveLopment Environment”, Master Thesis, CSE Dept., UNR, USA.

[10] Dascalu, S.M., Pasculescu, A., Woolever, J., Fritzinger, E., and Sharan, V. (2003) “Stratified Programming Integrated Development Environment (SPIDER)”, Proceedings of the 12th Intl. Conf. on Intelligent and Adaptive Systems and Software Engineering, 227-232.

[11] Westphal B, Harris, F., Dascalu, S., (2004). “Snippets: Support for Drag-and-Drop Programming in the Redwood Environment”, *Journal of Universal Computer Science*, 10 (7), 2004, 859-871.

[12] QT (2005). Trolltech: Cross-Platform C++ GUI Development and Embedded Systems Solutions. “Qt Application Framework” Accessed Jan. 28, 2005 at www.trolltech.com

[13] GTK+ (2005) GTK+: The GIMP Toolkit. Accessed Jan. 28, 2005 at http://www.gtk.org

[14] POSIX Threads (2005) “POSIX Threads Links.” Accessed April 2005 at www.humanfactor.com

[15] OpenSG (2005) OpenSG Home. Accessed March, 2005 at www.opensg.org

[16] ZThreads (2005) From Netinformations Computer Guide: Crahen, E. “ZThreads”. Accessed March, 2005 at zthread.sourceforge.net

[17] MsgConnect (2005) Eldos Corporation. “MSGConnect: There is a World to Connect”. Accessed March, 2005 at msgconnect.com

[18] HawkNL ™ (2005) Hawk Software. “HawkNL (Hawk Network Library)”. Accessed March, 2005 at www.hawksoft.com/hawknl

[19] UML (2005) OMG’s “UML Resource Page”. Accessed March 2005, at www.omg.org/uml

[20] Shultz, U. P., (2000) “Partial Evaluation for Class-Based Object-Oriented Languages”, Proceedings of the 2nd Symposium on Programs as Data Objects, 173-197.

[21] Gal, A., Wolfgang, S., and Spinczyk, O. (2001) “On Minimal Overhead Operating Systems and Aspect-Oriented Programming”, Proceedings of the 4th ECOOP Workshop on Object-Orientation.

[22] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns*, Addison-Wesley, 1994.