Type Laundering as a Software Design Pattern for Creating Hardware Abstraction Layers in C++

By

Cliff Michael McCollum
B.Sc., University of Victoria, 1996

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Supervisor: Dr. Hausi A. Müller

ABSTRACT

The concept of a hardware abstraction layer is a useful tool when designing software that must interface with third-party devices. The traditional approach to designing these abstraction layers in C++ provides weak isolation from the hardware and can make transitioning to new devices difficult. We show how the use of the Type Laundering software design pattern, coupled with an additional layer of logical indirection, can provide a much stronger degree of hardware isolation. We then provide examples of how this pattern has been used in an industrial-grade telecommunications system, and highlight some of the benefits and drawbacks discovered during the application of this pattern.

Table of Contents

List of Figures

List of Source Code Samples

Acknowledgments

This thesis is dedicated to my loving wife Deanna. Without her support I would not have completed it. It is also dedicated to my children: Zachary, Caesha, and Nyah. Their smiles and laughter are all the reward a father could ask for.

I am indebted to the help of many colleagues who assisted with the design and testing of the patterns presented in this paper. In particular, I would like to thank Steve Cockayne, Jason Corless, Greg Fox, and Rod Olafson. I would also like to thank my employer for allowing me to put effort into this thesis while continuing to work full-time.

I am also grateful for the members of the Rigi group at the University of Victoria, and for my Thesis Supervisor, Hausi Müller, whose ideas and suggestions have been invaluable.

1. Introduction

In the modern and highly competitive Telecommunications sector, many new systems are developed using off-the-shelf hardware components. This allows rapid system development and lower costs compared to solutions involving custom-designed hardware. Unfortunately, this method creates strong software dependencies on third-party components over which a designer often has little control. When creating a system with an expected lifetime measured in years, creating a dependency on any single component supplier is considered an undesirable business risk. Systems should be designed so that each component sourced from a third-party can be efficiently replaced with an equivalent component from another vendor. This reduces business risk if any component, or vendor, were to become unavailable or unacceptable in the future.

To design a system that utilizes numerous third-party components while retaining the flexibility to switch between components from various suppliers is a daunting challenge. This thesis presents a variation on the Type Laundering software design pattern. This pattern was created while the author was working on the design of a Carrier-grade Unified-Messaging system for the Telecommunications sector. We show how the Type Laundering pattern can be a useful tool when abstracting hardware interfaces away from application layer code.

1.1 Outline of Thesis

Chapter 2 provides some background information which puts this work in context: the Hardware/Software coupling problem, Software Design Patterns, a note on C++, and an example domain are all presented. Chapter 3 discusses the traditional approach to hardware abstraction, and presents the shortcomings of this method. Chapter 4 describes our solution in detail. Chapter 5 evaluates our solution compared to the traditional approach and provides examples showing how this pattern

has been useful in a production system. Chapter 6 highlights the contributions provided by this work. Chapter 7 concludes by summarizing our work and describing possible avenues for future research.

2. Background

This section discusses first, the basics of the hardware/software coupling problem. Secondly, it introduces the problem domain in which we developed our solution.

2.1 The Hardware/Software Coupling Problem

When integrating third-party components into a system you typically receive the *component* in three parts: the *hardware* itself, the hardware *drivers* that interface with the operating system, and the programming *libraries and headers* used to communicate with the drivers. It is the libraries and headers that form the *Application Programming Interface* (API) your code must manipulate to control the component. The API provided with the hardware can vary from low-level interfaces—similar to communicating with the hardware driver directly—to high-level libraries that can provide rich functionality to your application.

Regardless of the degree of utility presented in these libraries, the coding style and architecture required to make use of them are rarely identical to the styles and designs present in the rest of your application. This difference in style between third-party APIs and in-house code is the first challenge encountered when using third-party components. A common design technique is to create an abstraction layer to smooth out these differences.

This abstraction layer provides an interface that maps the third-party API to the existing coding structures and style in your application's own code (see Figure 2.1). An obvious example of this layer might be something that provides an object-oriented (OO) wrapper around a non object-oriented API.

When the decision is made to forgo an abstraction layer, the in-house code must interface directly to the third-party API. Because the third-party API is very solution specific, it frequently constrains the design of the in-house code. This might be the restriction to a particular programming method, or the requirement that all integers

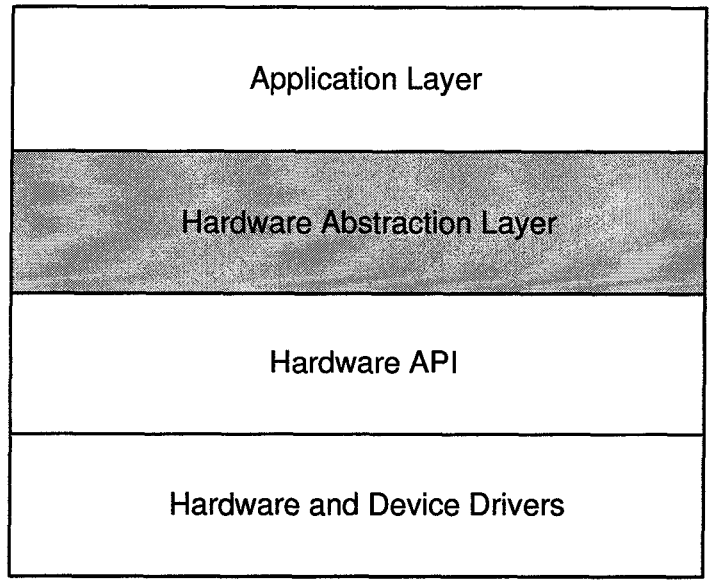

Figure 2.1: An Architecture With a Hardware Abstraction Layer

be represented in big-endian format, or any number of other possibilities.

Whether an abstraction layer is utilized or not, it is always necessary at some level in a design to have in-house code communicate directly with the third-party API. Where these two bodies of code interface is the source of the hardware/software coupling problem (see Figure 2.2). If the decision is ever made to change components, or if the third-party vendor were to change their API, it is the interface between the in-house code and the third-party API that would be forced to change. If this interface is well designed, the impact of such a change could be minimal. If this interface is poorly designed, an API change could affect every function in the product. Or worse yet, such a change might be impossible without re-writing the application completely.

There have been a number of mechanisms proposed for measuring the degree of coupling between software components in a given system. An obvious, but somewhat simple-minded technique, is to compare the total number of files that use third-party API functions or structures directly, against the total number of files in the application.

If the number of files directly using elements from the third-party API forms a sizable portion of the application, that system is said to have strong/hardware software coupling. In such a system developers would likely find moving to a new component or API difficult.

For some projects, however, this measure is clearly too simple. Consider a project involving ten thousand files that has only one percent of its files touching the third-party API directly. While this might be considered an example of low coupling, it still represents a project that would require the modification of one hundred source files if the third-party API changed. The requirement to change one hundred source files is enough to cause most designers to consider a system strongly coupled. As a result, it is also worth considering the absolute number of files that could be affected by a third-party API change and attempting to minimize this number.

In an object-oriented system, coupling analysis is often more complex. Factors such as inheritance and the use of private functions (such as the `friend` keyword in C++) can make the question more difficult. One proposal for analyzing coupling in object-oriented systems was made by Briand, Devanbu, and Melo in their paper "An Investigation into Coupling Measures in C++" [1]. Their method was further refined

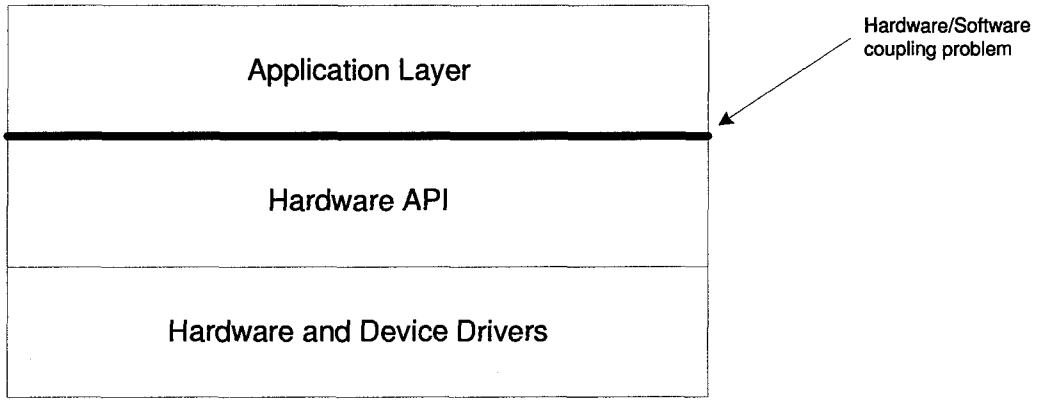

Figure 2.2: An Architecture Without a Hardware Abstraction Layer

in "A Unified Framework for Coupling Measurement in Object-Oriented Systems" [2]. Both of these papers attempt to take common object-oriented techniques into account when determining coupling metrics.

In our experience, most hardware-based API libraries are still not provided to developers with an object-oriented interface. As such, many of the more complex object-oriented analysis metrics are less applicable. When a design finally comes down to the point where a non-object-oriented hardware interface is being grafted onto an object-oriented hardware abstraction layer, it is often sufficient to consider a simple file-counting metric like the one described at the beginning of this section.

2.2 Software Design Patterns

This thesis is about design patterns. But what is a design pattern? Brad Appleton summarized design patterns as follows:

> "Fundamental to any science or engineering discipline is a common vocabulary for expressing its concepts, and a language for relating them together. The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions. Formally codifying these solutions and their relationships lets us successfully capture the body of knowledge which defines our understanding of good architectures that meet the needs of their users. Forming a common pattern language for conveying the structures and mechanisms of our architectures allows us to intelligibly reason about them. The primary focus is not so much on technology as it is on creating a culture to document and support sound engineering architecture and design." [3]

2.3 A Note on C++

The examples presented in this work are coded in C++, relying heavily upon object-oriented design techniques. While many design patterns can be implemented in non object-oriented languages, the pattern we present in this thesis relies on aspects of object-oriented programming for its functionality. While C++ is not the only object-oriented language we could have used, it is the easiest object-oriented language to interface with third-party devices (which usually provide C language libraries), and it is the language we used for the actual implementation and testing of this pattern.

While C++ is a very mature object-oriented language, it is important to note that it does not provide runtime reflection.[1] This lack of reflection works to the advantage of our pattern, ensuring it can properly hide the hardware layer. While this pattern can still be used in a language that provides reflection, such as Java, reflection capabilities can be used to work around, and thereby weaken, the abstractions provided by this pattern. The impact of reflection capabilities will not be considered in this thesis.

2.4 Example Domain

All the material presented in this work is drawn from a system developed while the author was working for a leading Telecommunications software provider. An explanation of the software system we developed, and the hardware and software used in that system, will make subsequent examples easier to understand. The system in question is a carrier-grade Unified Messaging and Communications system. It is capable of receiving and storing voice and fax messages; forwarding incoming calls to a list of destination numbers (the standard term for this feature is *Follow-Me*); providing notification of received messages via email or industry standard Message Waiting Indicators; and exchanging messages between messaging systems via Voice Profile for Internet Messaging (VPIM) messages. This system is fault-tolerant and fully redun-

[1] Reflection is the ability for objects to examine themselves at runtime—allowing a program to determine the types and functions each object supports.

dant, and is capable of supporting over six thousand simultaneous telephone calls and over ten million users.

2.4.1 Hardware

This system runs on industry standard PCI and compact-PCI (cPCI) chassis, using Intel Pentium processors. Telephony connectivity is provided via digital telephony cards from NMS Communications™ (NMS). In particular, this system has been designed around NMS's AG4000 [4] and CG6500C [5] products. The AG4000 is a high-density telephony card consisting of up to 48 Digital Signal Processing (DSP) cores, each capable of 100 Million Instructions Per Second (MIPS), a 100 MHz 386 compatible processor, and up to 4 T1 interfaces. The CG6500C is a higher-density card containing 96 DSP cores, a 500 MHz PowerPC processor, 16 T1/E1 interfaces, and two 100bT Ethernet interfaces (for Voice-Over-IP).

2.4.2 Software

The software was originally developed using C++, Java and an SQL database running on Windows 2000. In early 2004 everything was successfully ported to Linux, which now serves as the primary operating system for this product. The project consists of approximately one million lines of in-house source code. Control of the NMS hardware is performed using the Natural Access software provided by NMS.

Natural Access is the umbrella name for a collection of APIs that group common functions into *managers*. These managers include functions for voice play/record, call control and signaling, trunk monitoring, clock synchronization, call switching, multi-channel conferencing, fax sending/receiving, and tone detection/generation. The Natural Access API is a C-language interface where all functions are invoked by passing one or more opaque data-structures, originally obtained from Natural Access, along with the desired parameters. All functions are asynchronous, with control returning to the caller immediately after the function is invoked. Once the requested function has completed, an event containing the result is posted to a user-specified event queue.

Events are retrieved from this queue using the Natural Access `waitEvent` function, which blocks until an event is available. The Natural Access API includes more than one hundred unique data structures and hundreds of different functions.

3. The Traditional Solution

The first major risk involved in adopting any third-party component is the possibility that the vendor-provided API may undergo a substantial change after having been integrated into the system. In order to minimize the impact of this risk, it is common to provide an abstraction layer between in-house code and third-party APIs.

When using an object-oriented programming language, the most natural approach to this hardware abstraction layer is shown in Figure 3.1. The following features characterize this approach.

3.1 The Base Class Interface

Using standard object-oriented techniques, a designer first creates base classes that represent the generic interface to the hardware. This interface provides all the functions that would be expected in this type of hardware regardless of vendor. The intention of this interface is to provide a set of functions that will stay unchanged regardless of any future vendor replacement. In the case of a hypothetical hardware

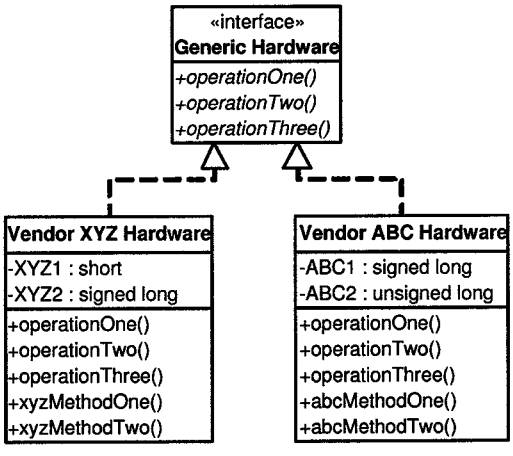

Figure 3.1: Traditional Solution

device, a simplified interface might look like the following:

Listing 3.1: An Interface in the Traditional Solution

```
// Abstract class to represent one endpoint of a phone call.
// All functions that concern a single phone call center around this class.
class CallEndpoint
{
  public:
  // Constructor/Destructor
  CallEndpoint(UInt8 boardNumber, UInt8 trunkNumber, UInt8 timeslotNumber)
     : iBoardNumber(boardNumber), iTrunkNumber(trunkNumber),
     iTimeslotNumber(timeslotNumber) {};

  virtual ~CallEndpoint();

  // Call Control
  virtual void answerCall(UInt8 afterThisManyRings) = 0;
  virtual void placeCall(string& numberToDial, UInt8 maxRings) = 0;
  virtual void disconnectCall() = 0;

  protected:
  UInt8 iBoardNumber;
  UInt8 iTrunkNumber;
  UInt8 iTimeslotNumber;
};

// Abstract class to represent the telephony hardware itself.
// Not the individual call endpoints.
class TelephonyHardware
{
  public:
  // Hardware Control
  virtual void startHardware() = 0;
  virtual void stopHardware() = 0;
};
```

The important point to observe is that these classes contain no functional code. They are, essentially, interfaces. As such, they represent an API that could be used with almost any type of telephony hardware. While this generic capability is desirable, these classes are obviously unable to get much done without vendor specific code. Somehow a way must be provided to control the particular hardware that is being used.

Furthermore, because of limitations in C++, the abstract class cannot specify everything we would like. For example, it is likely that all `CallEndpoints` will have methods to play and record sound files. Because these methods are likely to take parameters that are vendor specific (such as encoding definitions), these methods cannot be specified in a generic way.

3.2 Subclass for Specific Hardware

To support a particular choice of hardware, a subclass of the generic interface is created that adds code for each type of hardware supported. For example, to support hardware from vendor XYZ, the following code might be used:

Listing 3.2: Specific Hardware Subclasses

```
// Class to represent vendor XYZ's sound file. This class hides
// VendorXYZ API calls.
class XYZSoundFile
{
  public:
  // Construct a sound file
  XYZSoundFile(string& filename) : iFilename(filename),  iSoundHandle(0) {};

  // Open and close a sound file
  void openSoundFile() { iSoundHandle = VendorXYZOpenSoundFile(iFilename);  };
  void closeSoundFile() {  VendorXYZCloseSoundFile(iSoundHandle);  };

  // setters/getters
  UInt16 getSoundLengthInSeconds() {
     return VendorXYZSoundFileLengthInSeconds(iSoundHandle);
  };

  VendorXYZSoundHandle getXYZSoundHandle() { return iSoundHandle;  };

  private:
  string iFilename;
  VendorXYZSoundHandle iSoundHandle; // our sound file handle
};
```

```
// Class to represent XYZ's version of a phone call endpoint.
// This class hides VendorXYZ API calls.
class XYZCallEndpoint : public CallEndpoint
{
  public:
  // Construct/Destruct
  XYZCallEndpoint(UInt8 boardNumber, UInt8 trunkNumber,  UInt8 timesIotNumber)
     :   CallEndpoint(boardNumber, trunkNumber, timeslotNumber),
        iCallHandle(0) {
          iCallHandle = VendorXYZGetCallHandle(iBoardNumber,
           iTrunkNumber, iTimeslotNumber);
      };

  ~XYZCallEndpoint() { VendorXYZRelaseCallHandle(iCallHandle);  }

  // Call Control
  void answerCall(UInt8 afterThisManyRings) {
     VendorXYZAnswerCall(iCallHandle, afterThisManyRings);
  };

  void placeCall(string& numberToDial, UInt8 maxRings) {
     VendorXYZPlaceCall(iCallHandle, numberToDial, maxRings);
  };

  void disconnectCall() {  VendorXYZDisconnectCall(iCallHandle);  };

  // Voice play/record
  void playSound(XYZSoundFile& theFile) {
     VendorXYZPlaySound(iCallHandle, theFile.getXYZSoundHandle());
  };

  UInt16 recordSound(XYZSoundFile& theFile, UInt8 maxTime) {
     UInt16 length = VendorXYZRecordSound(iCallHandle,
        theFile.getXYZSoundHandle(), maxTime);
     return length;
  };

  private:
  VendorXYZCallHandle    iCallHandle;
};
```

```
// Class to control XYZ telephony hardware.
class XYZTelephonyHardware : public TelephonyHardware
{
  public:
  // Construct/Destruct
  XYZTelephonyHardware() : iBoardCount(0) { VendorXYZObtainHardware(); };

  ~XYZTelephonyHardware() { VendorXYZReleaseHardware(); };

  // Hardware Control
  void startHardware() {
     iBoardCount = VendorXYZGetBoardCount();
     for (UInt8 count = 0; count < iBoardCount; count++) {
        VendorXYZStartBoard(count);
     }
  };

  void stopHardware() {
     for (UInt8 count = 0; count < iBoardCount; count++) {
        VendorXYZStopBoard(count);
     }
  };

  private:
  UInt8 iBoardCount;
}
```

These classes add the XYZ specific API calls and data structures to the generic interfaces. A developer would directly create instances of these XYZ classes whenever an application requires communication with XYZ telephony hardware.

Using UML notation, this design looks like Figure 3.2.

3.3 Shortcomings of the Traditional Approach

While this approach is straightforward and easily implemented, it has a number of problems that limit its effectiveness.

3.3.1 Hard-Coded Vendor Classes

The most straightforward use of this pattern involves hard-coding a particular vendor's class names everywhere they are needed. For example, if it is known that

vendor XYZ is the preferred vendor at the moment, all uses of a `CallEndpoint` could be hard-coded to use the `XYZCallEndpoint` class. This technique is especially compelling given that the `XYZCallEndpoint` class has methods that take `XYZSoundFile` objects as parameters. By hard-coding the construction and parameter passing of XYZ objects, the code is easy to write and understand.

If vendor support was switched from XYZ to vendor ABC, all places where XYZ occurs in the code must be manually substituted with ABC. This can be done with a simple search-replace.

Alternatively, code for every vendor could use the exact same names, but be placed into vendor specific namespaces. This would lead to class definitions like:

```
namespace VendorXYZ
{
   class VendorCallEndpoint : public CallEndpoint {
      // code for Vendor XYZ's endpoint
   };
}
```

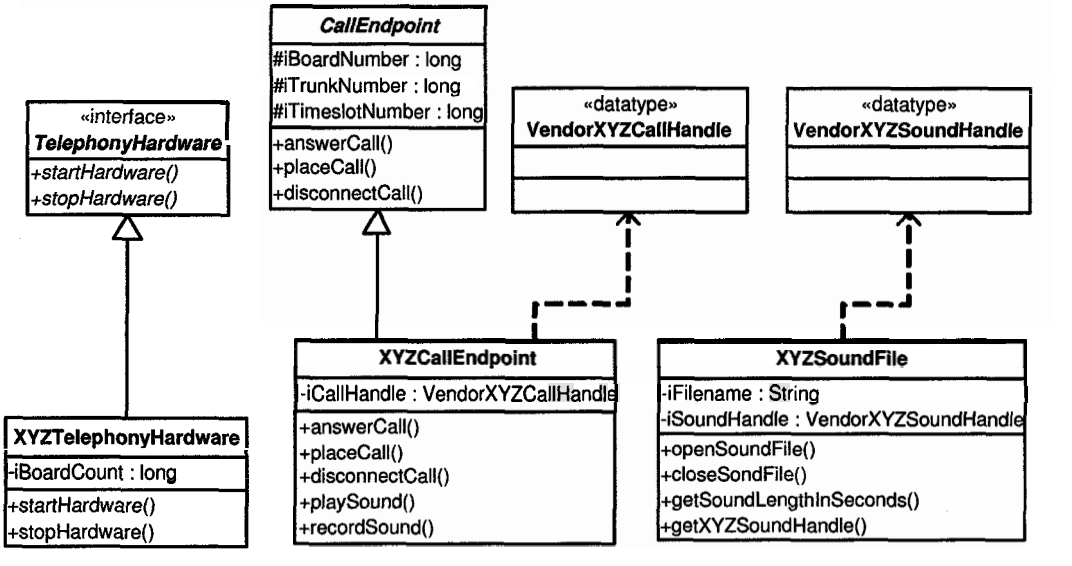

Figure 3.2: Example of the Traditional Solution

```
namespace VendorABC
{
   class VendorCallEndpoint : public CallEndpoint {
      // code for Vendor ABC's endpoint
   };
}
```

When namespaces are used, instead of performing a search-replace for a new vendor name, the following code could be inserted:

```
#define CurrentVendor VendorXYZ
```

Now all code can use statements like the following, and have them automatically substituted for the proper vendor whenever the `CurrentVendor` definition is changed:

```
// declare a call endpoint for the current vendor
CurrentVendor::VendorCallEndpoint ep(0, 0, 1);
```

If this method is used, only the `#define` statement needs changing if a new vendor is selected. This is certainly a substantial improvement over performing a search-replace on all the code. If this were the only problem we would have found an acceptable solution.

3.3.2 Determining Object Types

It is simple for a developer to build an XYZ object when it is known that XYZ hardware is in use. However, this places the onus on the developer to know, at some point within the code, which vendor's objects should be constructed. Furthermore, if multiple vendors are supported in the same code-base, the following becomes commonplace:

```
#if defined _USING_VENDOR_XYZ
XYZCallEndpoint* getNewEndpoint(UInt8 boardNumber, UInt8 trunkNumber,
    UInt8 timeslotNumber) {
    return new XYZCallEndpoint(boardNumber, trunkNumber, timeslotNumber);
}
#else if defined _USING_VENDOR_ABC
ABCCallEndpoint* getNewEndpoint(UInt8 boardNumber, UInt8 trunkNumber,
    UInt8 timeslotNumber) {
    return new ABCCallEndpoint(boardNumber, trunkNumber, timeslotNumber);
}
#endif
```

This coding style can quickly become cumbersome and difficult to read. The result is that many designers will place the in-house functions for each specific vendor class into its own file, and then use the following idiom to select the appropriate files at compile time:

```
#if defined _USING_VENDOR_XYZ
   #include "VendorXYZ_Functions.cpp"
#else if defined _USING_VENDOR_ABC
   #include "VendorABC_Functions.cpp"
#endif
```

While this approach simplifies reading the code, it does not change the essential fact that such a system is physically tied to a single vendor at compile-time.

3.3.3 Vendor Specific Methods

In addition to class naming issues, things are further complicated because the classes for each vendor's hardware may differ in significant ways. While the base class interfaces provide generic methods for most things, they cannot provide templates for methods that involve vendor-specific parameters, such as the `playSound` and `recordSound` methods in `XYZCallEndpoint` (see Listing 3.2). Neither can the generic interface provide templates for methods that are vendor-specific. In circumstances like these, the replacement of one vendor with another becomes much more complex than the simple name substitutions described in Sections 3.3.1 and 3.3.2.

3.3.4 Poor Information Hiding

One of the benefits of object-oriented design is its effectiveness at hiding non-public data.[2] In the case of hardware abstraction, the most obvious things to hide are those that pertain to the hardware itself: things which lie outside the scope of the generic interface. Unfortunately, in the Traditional Solution, the very things that we wish to hide are the same things that are made clearly visible in the hardware-specific files we require developers to include. By requiring the programmer to include the files for `XYZCallEndpoint` (for example), which files themselves include datatypes specific to vendor XYZ, we achieve the opposite of information hiding: instead of presenting the developer with less information when using our classes, we present them with more. Any solution that makes a developer explicitly aware that hardware specific files are being used, is a design that has failed to provide adequate information hiding in its hardware abstraction layer.

This information hiding failure leads to a secondary problem. Since developers must use the hardware-specific header files directly, they have clear view of all non-public data in those files, including things like private instance variables and protected methods. If the developers are well disciplined this may not be an issue. However, in our experience working with real developers, as soon as a deadline draws near it becomes difficult for developers to resist the temptation to use useful looking functions—even if they are in a non-public section of the class. Because the hardware specific files are likely included as source code, it is trivial for a developer to turn private methods into public ones and use them directly. If this were to occur, it would generate unexpected coupling in the code and make any future transition to an alternative hardware vendor dramatically more complex.

[2]This data hiding is often referred to as *encapsulation* and is not exclusive to object-oriented languages. Many other languages, such as Module-2 and Ada, provide similar features.

4. Our Solution

This section describes the design pattern we created for hardware abstraction. It briefly describes some of the investigation that lead to our solution. As well, it presents some sample code showing this pattern in use, and discusses how this pattern can be used in practice. A formal structure diagram of this pattern can be found in Section 4.7.

4.1 Requirements

The early versions of our software system used a hardware abstraction mechanism similar to the traditional approach. This design allowed us to gain experience in the problem domain (namely, telephony and Telecommunications hardware) and to gain grounding in the general principles behind the hardware we were using. It was also a time during which we analyzed alternative hardware solutions and familiarized ourselves with the differences between them.

4.1.1 Problems with Our Original Design

After evolving our original design for nearly two years, it became clear that there were some shortcomings we needed to address. In particular, we noticed:

- There was very little abstraction in our middle layer. New high-level behavioural requirements always prompted changes in both the middle and bottom layers.
- Our middle layer was using vendor specific structures in its public API. This forced the use of vendor specific code in our application layer.
- Our application layer was dependent upon state machines that existed at the vendor API level. If the behaviour of a particular vendor's state-machine changed, our application code would break.

4.1.2 Guiding Principles

When considering alternatives to our existing design, a few important principles emerged:

- We wanted to ensure that changes could occur at the hardware level (either a new vendor or changes to an existing vendor's API) without requiring code changes at the application level.

- No vendor specific information should be visible to the application layer. While this obviously included data structures and function names, we also wanted it to include the vendor choice itself. The idea was to ensure that no code outside the hardware abstraction could determine which brand of hardware was in use. This was considered important in order to prevent the application layer from acquiring unwanted dependencies upon particular hardware devices.

In addition to these general principles, some specific requirements related to C++ were selected:

- Application layer code should not have to include hardware specific header files for any reason.

- Application layer code should not be aware of any hardware specific functions or data structures—even if hardware specific classes are being used. In other words, hardware specific classes, if they are used directly, must have two interfaces: there should be a narrow interface that presents the application layer with generic hardware functions only, and there should be a wide interface that makes vendor specific functions available to the abstraction layer itself, but does not make those functions visible outside the abstraction layer. Effectively, we wanted classes that looked like Figure 4.1.

- The public API of our abstraction layer should be rich enough to support hardware from any vendor. If a specific vendor did not support a feature available in

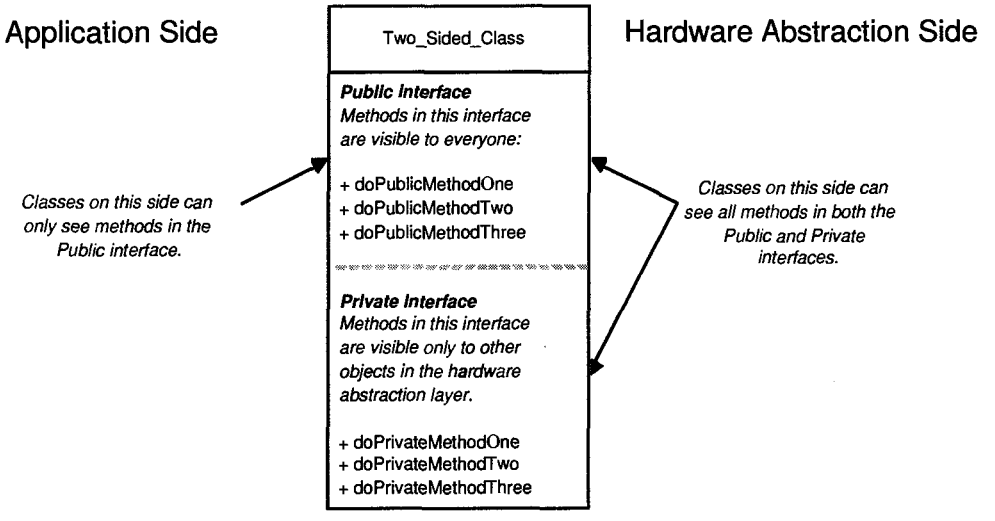

Figure 4.1: Two Sided Class

the abstraction layer, only that missing feature should be unavailable. All other features should still function. Non-supported features should fail at compile-time where possible; where this is not possible, these failures should happen through C++ exceptions at run-time.

4.2 Examining Other Solutions

Before spending time on a solution we decided to examine other hardware abstraction layers to learn how other designers had tackled various aspects of this problem. Our examination included such well-known software abstraction layers as the Java Abstract Windowing Toolkit (AWT), and the Microsoft Telephony API (TAPI). We also examined the APIs from three major telephony hardware vendors—Brooktrout, Dialogic, and NMS—to understand what sorts of situations we might have to accommodate. Each solution we investigated presented a wealth of useful ideas.

4.2.1 Java Abstract Windowing Toolkit

The Java AWT is an obsolete abstraction layer as of 2004. However, when we first began looking at it in 1999, it was considered in its prime and was just being replaced by the more sophisticated SWING classes. The AWT was designed to abstract the Graphical User Interface (GUI) of each operating system the Java platform supported. It provides functions for window handling, widget creation, menus, and all the other traditional GUI elements. While this may not seem like an obvious candidate when considering telephony hardware abstractions, its inclusion was deliberate.

The primary reason we investigated the AWT was its effectiveness at hiding a software layer as complex as the Windows or Macintosh GUI APIs. While a GUI is a quite different from telephony hardware, they have many items in common. They are both event driven (that is, things can happen due to forces outside the application layer's control), and they both group their functions into sets (such as menuing, windowing, and fonts in a GUI).

One of the most useful ideas we extracted from the Java AWT was the notion of a *Peer* class. In the AWT, the application layer communicates with a particular AWT class, such as the Windowing class. This class would contain the public API for windowing, while making no OS/GUI specific functions available. The AWT Windowing class would, in turn, communicate with a Windowing Peer class. The Peer class was responsible for translating the Java generic windowing requests into OS/GUI specific API requests. This design is shown in Figure 4.2. Readers who are familiar with other software design patterns will recognize this as an implementation of the Bridge pattern.

The key to this design is that the application layer does not communicate directly with a subclass of some generic interface (as in the traditional approach). Rather, the application layer communicates with a generic class directly. The generic class then communicates with an OS/GUI specific Peer class that has a mechanism for translating each request into hardware specific API calls. It is up to each Java

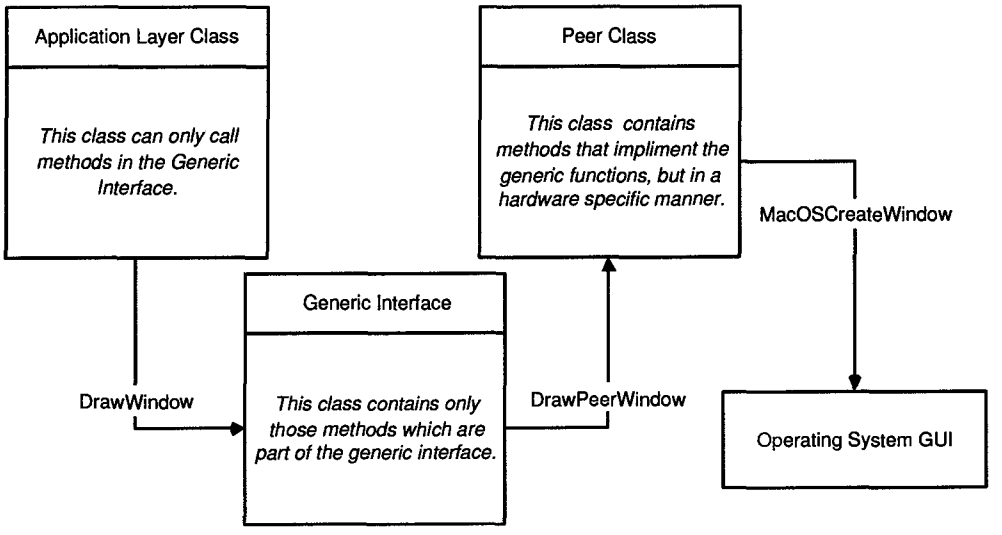

Figure 4.2: Java Peer Class

AWT implementation to ensure that OS appropriate Peer classes are made available to the AWT without the application layer's intervention.

The idea of hardware specific peer code, which is not visible to the application layer, was an idea we wanted in our design.

4.2.2 Microsoft's Telephony API

This is Microsoft's official abstraction layer for the voice and fax features of telephone based devices. While the Microsoft Telephony API (TAPI) is designed for much smaller hardware devices than we would be using (TAPI is intended for one or two simultaneous connections, while we would be supporting hundreds or thousands), it does contain a few important ideas.

In TAPI nearly all functions are asynchronous, with results returned through a callback function registered with the API. While some telephony hardware vendors followed this pattern, others returned function results in an event queue. It was determined that we would adopt a callback based mechanism similar to TAPI's. However, unlike TAPI, which has a single callback function for all results, we wanted to support

multiple callback functions—one for each category of telephony functions we would support (fax, call control, etc). The decision to support multiple callback functions was made so the application layer could more easily divide logical responsibility for portions of a call. Our earlier designs had made this type of logical division difficult and we wanted to avoid repeating that mistake.

4.3 The History of Type Laundering

By combining many of the ideas we had seen in other solutions, and by adding a few of our own, we came up with an approach to hardware abstraction that we considered very novel at the time. The basics of this pattern were documented in our offices in the winter of 1999 and first used in our code in the spring of that year. It was much later, around the Fall of 2001, that we discovered a similar pattern had been published by John Vlissides in *C++ Report* in February of 1997 [6] and later in his book *Pattern Hatching* [7]. While we were unaware of that paper during the design of our solution, its general principles are identical to our own.

An important difference between Vlissides' solution and our own, however, centers around the intent of the pattern. Vlissides presented his pattern as a solution to the problem faced by Framework developers who wish to provide a set of core functionality, while allowing application developers to enhance the framework in an extensible fashion. In short, Vlissides was seeking a pattern that allowed application layer code to *add* functionality to a lower-level framework layer, without getting caught up in the mess of type-casting that often results in such a scenario.

Our solution, on the other hand, was a pattern designed to allow lower-level code to *hide* functionality from the application layer, while maintaining the flexibility to expand those lower layers without affecting the design and structure of the layers above it. In effect, we were trying to solve the opposite problem that Vlissides cited as his motivation.

As it turns out, the patterns that we and Vlissides came up with rely upon an identical mechanism—though we use it in a nearly opposite manner. Vlissides

named his pattern *Type Laundering.* We believe this name has reference to "money laundering", which is the idea of purposefully hiding a source of illegitimate income; in the case of software development, the phrase has reference to the idea of purposefully hiding the type of a returned object. Because Vlissides' publication appeared before ours, we have elected to honour the name he gave to it, and present our pattern as a variation on Type Laundering. While we cannot claim to have created this pattern, we do believe that our application of this pattern is unique.

4.4 Type Laundering Explained

To save the reader some effort looking for Vlissides' Type Laundering paper, this section contains a summary of his original presentation on the subject. For a more thorough description please see the original publication [7].

4.4.1 Vlissides Original Concept

Imagine a project in which you are using a third-party provided framework. You have extended many of this framework's interfaces to add the functionality you require. Unfortunately, this framework cannot possibly know about the extensions you have created, and you find yourself having to use `dynamic_cast` everywhere in order to recover your extended types. There must be a way around this.

As a concrete example, consider a framework that provides support for real-time control of embedded devices. This framework defines an abstract base class `Event`. To use the framework you subclass `Event` to provide your domain-specific events. The framework designers have provided only the basics in their interface. In fact, `Event` only defines a couple of operations that would be applicable for all event types:

```
virtual long timestamp() = 0;
virtual const char* rep() = 0;
```

where `timestamp` defines the precise time of an event's occurrence, and `rep` returns a low-level representation of the event - probably a string of bytes straight from the device under control. It becomes the job of each subclass to define more specific and

useful features from these two basic operations.

Vlissides proposed a vending machine in his example. This machine has a `CoinInsertedEvent` subclass that adds a `Cents getCoin()` operation. This operation returns the value of a coin inserted by a customer. There is another event, the `CoinReleaseEvent`, that gets created when a customer requests their money back. These events will have to be implemented using the `rep` data. As Vlissides points out, callers could use the `rep` data directly (if it is public), but there would be little point because it offers almost no abstraction and will be difficult to use.

Of course, there is a basic problem here, and it stems from the framework's narrow event interface. The framework cannot know about any domain-specific events, since those were not envisioned until after the framework was complete. The `timestamp` and `rep` operations are all the framework has to offer.

This raises two questions:

1. How does the framework *create instances* of domain-specific subclasses?

2. How does application code *access subclass-specific operations* when all it gets from the framework is objects of type Event?

Vlissides answers the question about object-creation by referring to other well-known software patterns. He indicates that both the Factory and the Prototype pattern offer good solutions to this dilemma. We will not dwell on this aspect of his paper because it is not important to our discussion.

However, Vlissides' second question is important to us: are there patterns for recovering type information from an instance? More specifically, if the framework provides an operation like

```
virtual Event* nextEvent();
```

how does the application know which kind of event it gets so that it can call the right subclass-specific operations?

Of course, there is the obvious brute-force approach:

```
Event* e = nextEvent();
CoinInsertedEvent* ie;
CoinReleaseEvent* re;
// similar declarations for other kinds of events

if (ie = dynamic_cast<CoinInsertedEvent*>(e)) {
    // call CoinInsertedEvent-specific operations on ie

} else if (re = dynamic_cast<CoinReleaseEvent*>(e)) {
    // call CoinReleaseEvent-specific operations on re

} else if (...) {
    // ...you get the idea
}
```

We would not have to put this code in too many places before it started to become painful. In addition, this only becomes worse each time we add a new event type. We would like to find a better solution.

At this point in his discussion, Vlissides describes the Memento pattern and how it relates to this problem. Memento's purpose is to capture and externalize an object's state so it can be restored at a later time. Plus, Memento must do this without violating that object's encapsulation. The implementation of Memento often involves the use of the `friend` keyword. However, many designers prefer to avoid use of the `friend` keyword whenever possible. In addition, `friend` is not inherited by subclasses and this can create problems of its own.

Here Vlissides introduces his new pattern and gives it the name *Type Laundering.* His idea is to start with an abstract base class that includes only the elements of its interface that should be public, which in the case of this example is only a destructor.

```
class Event {
  public:
    virtual ~Event () { }
  protected:
    Event () { }
    Event& operator= (const Event&) {}
};
```

The default and copy constructors are protected in the example to preclude instantiation—that is, to ensure `Event` acts as an abstract class. Given this base class, a `CoinInsertedEvent` subclass of `Event` might be written as:

```
class CoinInsertedEvent : public Event {
  public:
    CoinInsertedEvent () { iType = tNullCoin; }

    CoinType getCoinType () const { return iType; }
    void setCoinType (CoinType t) { iType = t; }
  private:
    CoinType iType;
};
```

This arrangement requires all code that cooperates with our new `Event` subclass to downcast `Event` objects to a `CoinInsertedEvent` before they can access the new interface. Inside `EventHandler` this may look like:

```
class EventHandler {
    // ...

    virtual void receivedCoin (Event& e) {
          CoinInsertedEvent* ce;

        if (ce = dynamic_cast<CoinInsertedEvent*>(&e)) {
            ce->setCoinType(newCoinType);
            // newCoinType is the type of coin just
            // inserted - which EventHandler keeps internally
        }
    }

    // ...
};
```

The dynamic cast ensures that the `receivedCoin` method will access and modify only `CoinInsertedEvent` objects and ignore everything else.

As final clarification on this pattern, Vlissides discusses how events get instantiated. Clearly, clients are in a quandary because they can no longer instantiate an `Event` object or its subclasses directly. While nothing actually prevents the client from creating these objects, it is impossible know which type of object each `EventHandler` requires.

To solve this dilemma, Vlissides proposes a variation on the Factory Method to provide for abstract instantiation:

```
class EventHandler {
  public:
    // ...

    virtual Event* event () { return new CoinInsertedEvent; }

    // ...
};
```

Because `event()` returns something of type `Event*`, clients can't access the subclass-specific operations unless they start dynamic-casting randomly to figure out the type—and even that's not an option if `CoinInsertedEvent` is not exported in a header file.

4.5 A Type Laundering Variation

Now that we have explained Vlissides' concept of this pattern, we describe our variation on Type Laundering and show how it can be used as a tool for hardware abstraction.

A simple block diagram outlining our design is shown in Figure 4.3.

4.5.1 Function Kits

The first decision we made when redesigning our hardware layer was to group all the functions we would need into sets of related behaviours based upon the problem domain. This grouping allowed us to concentrate our design on a narrow set of methods with a small set of shared data structures. We called each collection of related functions a *kit*. A class was created for each kit which acted as the public interface for that kit's functions. In effect, the kit was expected to operate like the traditional *Adapter* pattern [8]. However, as the design grew, and additional logic was placed into the kits, they begin taking on aspects of the *Mediator* pattern [8] as well.

Because our particular problem was related to telephony, our final design had kits for *Call Control, Tone Detection and Generation, Fax Sending and Receiving, Call Switching*, and *Voice Playback and Recording.* As an example, the class representing our Call Control Kit is shown below.

Listing 4.1: Call Control Kit

```
// Kit to provide telephony call-control functions.
class CallControlKit
{
  public:
   // Constructor and Destructor
    CallControlKit(CallEndpoint* ownerEndpoint);
    virtual ~CallControlKit();

    // Return the endpoint associated with this kit
    CallEndpoint* getEndpoint();

    // Answer an incoming call
    virtual void answerCall(UINT32 ringCount);
```

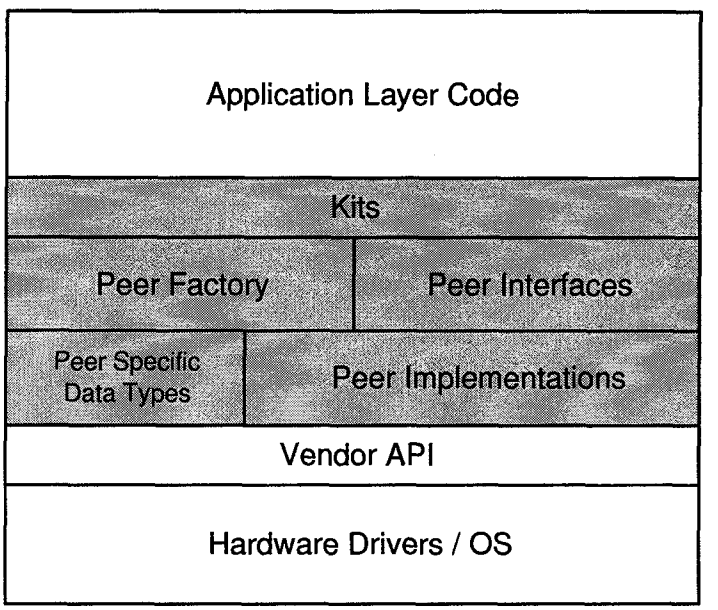

Figure 4.3: Block Diagram of the Kit and Peer Classes

```
    // Release an incoming call
    virtual void releaseCall();

    // Place an outgoing call.
    virtual void placeCall(PhoneNumber& destNumber, PhoneNumber& origNumber,
      UINT32 ringCount = 7);

  private:
    CallEndpoint* iEndpoint;    // Call endpoint for this kit
    CallControlPeer* iCCPeer; // Peer to provide Call Control
};
```

In addition to providing the public interface for a common set of functions, each kit provides an added layer of indirection. The kit makes it possible to have extra state and data validation between the higher layer application code and the lower level hardware abstraction layer. As a system grows in complexity, the kit abstraction can become invaluable.

Our design stated that each kit should not perform any hardware interaction directly. Instead, that responsibility is delegated to a *peer* object held by each kit (see Section 4.5.2). This decision was made to clearly separate the logical abstraction capabilities provided by the kit object from the hardware functions provided by each peer.

An example of this delegation can be seen in the Call Control Kit's `releaseCall` method:

```
void CallControlKit::releaseCall()
{
  if(releaseIsValid()) { iCCPeer->releaseCall(iEndpoint);  }
}
```

In this listing it is easy to see the general pattern for all kit functions. When a function is called, the kit performs state checking (or any other behaviour deemed important) and then delegates the hardware function to its peer object. By keeping its responsibilities limited to state validation manipulation, the kit's abstraction stays clean and clearly defined.

4.5.2 Peer Objects

Each kit contains a pointer to its *peer* object. This peer is responsible for interacting with the vendor provided API. An example of the Call Control peer object is shown below:

Listing 4.2: Call Control Peer

```
class CallControlPeer
{
  public:
    // Default destructor
    virtual ~CallControlPeer();

    // Answer an incoming call
    virtual void answerCall(CallEndpoint* endpoint, UINT32 ringCount) = 0;

    // Release a disconnected call
    virtual void releaseCall(CallEndpoint* endpoint) = 0;

    // Place an outgoing call
    virtual void placeCall(CallEndpoint* endpoint, PhoneNumber& destNumber,
     PhoneNumber& origNumber, UINT32 ringCount) = 0;
};
```

A quick glance at this code makes it clear that this is a C++ abstract class and acts only as an interface definition. It is up to a subclass to provide the concrete implementation for hardware-specific functions and data types. Obviously, there would be one subclass for each hardware API supported by the system.

The division of responsibilities between a Kit and a Peer creates a challenge. The design calls for the kit to have no vendor specific code, while at the same time delegating hardware requests to a vendor specific subclass of the Peer interface. How can the kit obtain a reference to a hardware-specific subclass without containing the code necessary to build such a class? We clearly could not have application-layer code building peer objects because that would violate the entire intention of an abstraction layer. We determined that we would have to construct the peer object at a code layer below the kit without the kit objects knowing how this was happening.

Enter the *Factory* design pattern.

4.5.3 The Object Factory

The Factory pattern is one of the patterns presented in the original *Design Patterns* [8] book. The idea behind this pattern is to use an object of one type as a *factory* to produce objects of another type. We realized that this was the key we needed. We could produce a *peer factory* specifically to create the appropriate subclasses of our peer interfaces, and have it return those subclasses using the base class as its return type. In its most basic form, this concept is expressed in code as:

Listing 4.3: Type Laundering with a Shape Factory

```
class Shape {
   public:
      Shape();
      doSomething() = 0;
};

class Square : public Shape {
   public:
      Square();
      doSomething() { std::cout << "hello" << std::endl; }
      doSomethingElse() { std::cout << "goodbye" << std::endl; }
};

class ShapeFactory {
   public:
      static Shape* getShape() { return new Square(); }
};
```

Notice that the `ShapeFactory` object creates instances of the `Square` class, but returns them as pointers to the base `Shape`. In this way, callers that request a shape from the factory are unaware that they have been returned a square. It is this idea that makes the *double-sided class* possible (see Figure 4.1). In effect, the object returned from the Factory is considered a Shape by the caller, but internally it is a Square and can access all of the extra functions unique to a Square. The only way for a caller to determine whether the Shape they have received is indeed a Square (or some other object) would be to start using the `dynamic_cast` operator on random class names until one of the casts succeeds.

This mechanism provides an effective way to produce objects that contain hardware-specific functionality, without making the construction of those objects visible to higher layers. All the caller has access to is the base class portion of the interface, and the methods and data types that it contains. Any extensions to this interface are forever hidden from the caller's view.

By using this pattern, it becomes possible for a hardware abstraction layer to return instances of hardware-specific classes, without the kit or application layer knowing which subclasses they are dealing with. In fact, the application layer is completely unaware that any subclassing has even taken place. The higher layers only need access to the interface files for the base class and for the object factory. Application developers are freed entirely from the need to use hardware-specific files, data structures, objects, or any other vendor specific items.

These factory generated objects are then used by calling the methods made available through the abstract base class. When such a call is made, the dynamic type of the object is determined at runtime and the proper function in the hardware-specific object is called.

4.5.4 Hiding Data Types

If hardware-specific subclasses were limited to only those methods described in their abstract base class, it is unlikely that they could accomplish very much. Fortunately, the hardware-specific subclasses are free to expand their interfaces without worrying that their interfaces will become visible to application-layer code.

Once the dynamic type of an object is determined by the runtime system, and the virtual function call has been dispatched, program execution enters the hardware-specific subclass. Once inside the subclass method, it is possible to make use of additional hardware-specific methods that are not publicly available in the base class interface. For example, once inside the `doSomething` method of the `Square` class, it becomes possible to make calls to the `doSomethingElse` method. While this expands the functions available to subclasses, it still does not offer enough flexibility. Without

hardware-specific data types, this layer is still very restricted.

How does a hardware-specific subclass pass hardware-specific data structures in and out of the application layer code? How can it do this without the application layer being aware that it is dealing with hardware-specific types? Like most things in computer science, the answer is to add another layer of indirection. In this case, the indirection consists of wrapping vendor-specific data structures in a pattern identical to the one we are already using for vendor-specific functions. Consider this extended version of the Shape example:

Listing 4.4: Using a Data Structure Factory

```
class ShapeData {
   public:
      ShapeData(int param1) = 0;
   protected:
      int data;
}

class SquareData : public ShapeData {
   public:
      SquareData(int param1) {
         data = param1;
         sqData = data * 2;
      }

      int getData() { return data; }
      int getSquareData() { return sqData; }

   protected:
      int sqData;
}

class ShapeDataFactory {
   public:
      static ShapeData* getShapeData(int param) {
         return new SquareData(param);
      }
};
```

```
class Shape {
   public:
      Shape();
      doSomething(ShapeData& data) = 0;
};

class Square : public Shape {
   public:
      Square();
      doSomething(ShapeData& data) {
         SquareData& sqData = dynamic_cast<SquareData&>(data);
         std::cout << "Shape:" << sqData.getData() << std::endl;
         std::cout << "Square:" << sqData.getSquareData() << std::endl;
      }
};

class ShapeFactory {
   public:
      static Shape* getShape() { return new Square();  }
};
```

In this example, the pattern shown in the Shape objects is repeated in the ShapeData classes. With a structure like this, the following code becomes quite interesting:

```
void interesting() {
   ShapeData aData = ShapeDataFactory.getShapeData(1);
   Shape theShape = ShapeFactory.getShape();
   aShape.doSomething(aData);
}
```

What happens when we execute this code? First, the ShapeDataFactory returns a SquareData object, initialized with the value of 1. The application code is only aware that this is a ShapeData object, and does not have access to any methods that allow manipulating this as a SquareData object. Next, the ShapeFactory is called as before, and a Square object is returned. Again, the application code is only aware of this object as a generic Shape. Finally, the call to doSomething is made. At runtime, the system determines that theShape is actually a Square and calls the doSomething method in the Square class.

Inside this method, the parameter is passed as the generic `ShapeData` type. However, with the help of `dynamic_cast`, the code is able to safely recover the dynamic type of this object and turn it back into the `SquareData` that is really is. Once this conversion has been made, the `Square` object has full access to the `SquareData` class and any hardware-specific methods and data it may contain.

If a system design incorporates hardware abstractions for multiple vendors at the same time, it becomes possible for a programmer to pass a data object designed for one vendor to a method designed for another vendor's hardware. For example, an extension of the listing above could also return objects of type `CircleData`. If such an object were passed to the `doSomething` method of a `Square` incorrect behaviour is likely to result. In this case, the return value of the `dynamic_cast` should be checked against `null`. Failures to perform the conversion should be identified as a runtime error.

This pattern can be extended and repeated as often as required to hide any number of hardware-specific objects and data.

4.6 The Factory Problem

So far we have presented a pattern that allows very flexible hiding of hardware-specific classes and data. However, this pattern is dependent upon intelligent `factory` classes that are capable of building appropriate subclasses of objects for the hardware being used. So long as these factories are in place, we can switch to different subclasses in order to support an alternative hardware vendor, and no application-layer code needs to change. But what about the factory objects themselves?

Re-coding the factories to switch to an alternative vendor is a simple task: we only need to change the class names used in the `new` method calls. This change will only affect a few lines of code in each factory. Since the number of factory objects is likely to be reasonably small, this can be considered a trivial modification.

Is it possible to make this better? Certainly. The first improvement might be the creation of a single factory responsible for constructing all hardware objects in

the system. This way we can limit the changes to a single file when we switched from one vendor to another. However, an even better solution would provide a mechanism that could detect the installed hardware at runtime and construct the appropriate factories on demand. This way we would never need to change anything when we ran on hardware from different vendors. While this second solution is possible, it is not considered in this thesis.

4.7 Pattern Documentation

This section documents the Type Laundering pattern as it is used for hardware abstraction. We have tried to adhere to the Gamma *et al.* style [8] as closely as possible. Some sections in this pattern documentation have been substantially shortened because their information is available elsewhere in this thesis.

Pattern Name Type Laundering Variation for Hardware Abstraction

Classification Class Behavioral

Intent Provide a mechanism to hide hardware abstractions and allow those abstractions to be modified and expanded without affecting client code.

Motivation A hardware abstraction layer is traditionally constructed in a manner that makes migrating to different hardware or APIs difficult. This pattern provides a mechanism for constructing a hardware abstraction layer that removes most of the problems that plague the traditional approach to this problem.

Applicability Apply Type Laundering when *all* of the following are true:

- You have a set of API functions you wish to completely isolate from client code.
- You are using an object oriented language that has support for runtime type detection (like the dynamic_cast function in C++). This pattern is less applicable for languages with reflection capabilities (like Java).

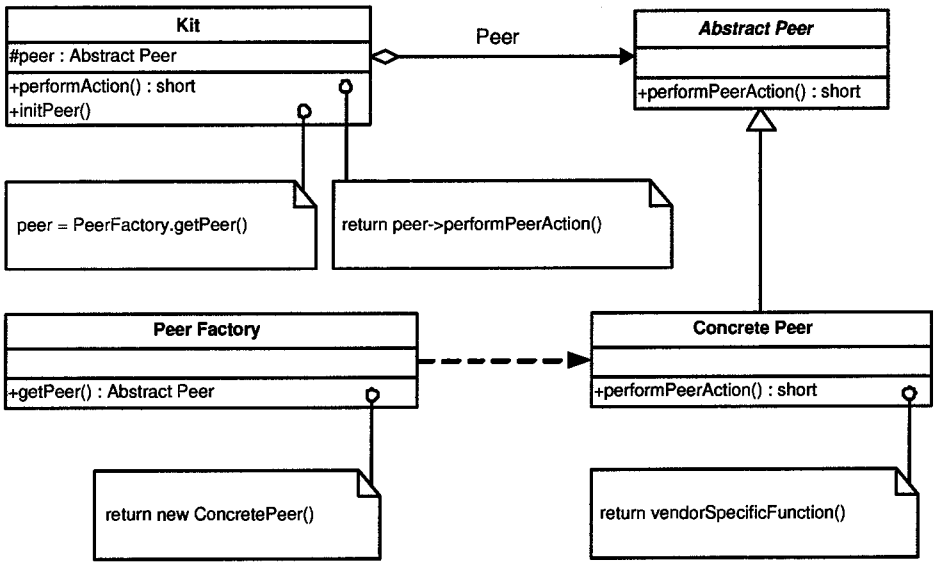

Figure 4.4: Type Laundering Variation Structure

Participants

- **Abstract Peer**
  - Provides interface to Kit for manipulating generic version of hardware or API.
  - Provides abstract base class for Concrete Peers that interact directly with hardware or API.
- **Concrete Peer**
  - Provides concrete implementation of Abstract Peer.
  - Directly interacts with hardware or API being abstracted.
- **Peer Factory**
  - Creates instances of Concrete Peer that are appropriate for the type of hardware or API being used.

  - Returns created instances using Abstract Peer type.

- **Kit**

  - Provides the public interface to the hardware abstraction layer.
  - Obtains a pointer to an Abstract Peer from the Peer Factory.
  - Delegates function requests to Abstract Peer pointer (which is an instance of Concrete Peer).

Collaborations

- Concrete Peer implements the Abstract Peer interface.
- Kit delegates functionality to subclasses of Abstract Peer to allow manipulation of hardware or API.
- Peer Factory creates subclasses of Abstract Peer for Kits.

Consequences Type Laundering provides the following benefits:

1. *Vendor specific information is decoupled from client code.* All vendor specific knowledge is hidden inside Concrete Peers.
2. *Responsibility for constructing appropriate hardware objects is hidden inside the Peer Factory.* Client code does not need to know what type of hardware is being used and what objects are interacting with it.
3. *Vendor specific function calls and logic are hidden behind two layer of abstraction.* The Kit and Peer classes isolate all vendor specific logic and function calls from clients.
4. *Vendor specific header files and data structures remain hidden.* Header files required to use a vendor API are restricted to the Concrete Peer objects. Vendor specific data structures can be abstracted behind another instance of the Type Laundering pattern.

There are three main liabilities to this pattern:

1. *Double the number of classes.* Instead of a traditional base class and its subclass, this pattern requires the addition of the Peer Factory and Kit classes.

2. *Memory management is complicated.* Because Concrete Peer objects are returned as pointers from the Peer Factory, extra care must be taken to ensure their correct cleanup.

3. *Peer Factories must still be hardware specific.* Somewhere, there must still be code that knows which type of hardware is being used. While this pattern moves that knowledge deep inside the abstraction layer, it cannot remove it completely. If new hardware is supported, the Peer Factory objects will have to be managed to ensure the appropriate types of Concrete Peer objects are created.

Implementation See Section 4.5.

Sample Code See Section 4.5.

Known Uses See Section 5.

5. Evaluation

The previous section presented the *Type Laundering* pattern and showed how it can be used to effectively hide hardware specific subclasses from application layer code. This section presents practical examples showing how Type Laundering has been used as part of a large commercial application. This section also considers alternatives to this pattern and how they compare with Type Laundering.

5.1 The Pattern in Practice

Before we began using the Type Laundering pattern, we maintained a large body of code utilizing a traditional approach to hardware abstraction. This involved a collection of objects that were written specifically for one hardware vendor and were used directly in our code. In addition, we had numerous references to vendor provided header files and data structures throughout the lower layers of our application. To be truthful, this approach was working fine for us. However, we recognized that it was important to insulate our business from the risk associated with using a single hardware vendor. This necessitated changes to our code to make it hardware and vendor agnostic.

The changes needed to create hardware independence in our code would be substantial. At the same time we were aware of other portions of our application that were ready for full or partial re-factoring. Seizing this opportunity, we proposed a complete rewrite of our application. Management accepted our proposal, and we began looking for a way to solve the hardware abstraction problem. Section 4.2 discusses some of what we considered as we worked on the problem.

Because our code underwent a complete rewrite as we adopted Type Laundering, we do not have any experiences related to taking a traditional hardware abstraction layer and transforming it to a Type Laundering based solution. However, it is our opinion that a transition of this type is unlikely: most abstraction layers using the

ideas in this thesis will either be completely rewritten to make use of these ideas, or they will be part of a new project with no existing code base.

5.2 Designing with the Pattern

While we cannot comment on the process of migrating an existing design to a Type Laundering solution, we can make a number of observations about this pattern as it was used during a complete re-write of our messaging product.

One of the prime motivations for rewriting our product in 1999 was to reduce our dependence upon a single hardware vendor. Because we wanted to reach a point of stability with our hardware abstraction interface as quickly as possible, our design work started by creating the various *kits* described in Section 4.5.1. Once these kits were in place, it was possible to begin rewriting our core application logic against these new APIs even though the hardware integration was still a work in progress.

The next step was to create the base class *peer* objects called by the kits (see Section 4.5.2), and then to create the peer subclasses to control the hardware we were using. Finally, we created a set of *factory* objects to handle the type-laundering of these peers on behalf of our kits (see Section 4.5.3).

This design and prototyping process took us approximately one month—excluding the time spent devising the Type Laundering pattern itself. While there were continued adjustments that needed to be made, the code was complete and functional in under two months. This duration compared very favorably with the time spent creating the previous, traditional style, hardware interface layer. While this new body of code was certainly larger and more complex due to its additional abstractions, we were also more experienced with the problem domain and had our previous experience to draw upon.

5.3 Working with the Pattern

Once our Type Laundering solution was in place, we began replacing the core application logic that utilized this new hardware abstraction layer. All the developers

involved in this project agreed that these new classes were no harder to use than the ones they replaced. In some cases, the new classes were easier to use because they hid the vendor API completely, presenting a more consistent set of data structures. In addition, the *kit* layer often made design easier because it presented a better logical abstraction than our previous design.

The total time required to rewrite the hardware abstraction layer and the accompanying application layer code was approximately six months. However, this time also included re-writing many additional portions of the product—including the entire database abstraction layer.

Once this new pattern had been put into place and used for close to a year, the first real opportunity to test its power was encountered.

5.3.1 API Migration

One of the inevitable consequences of the endless march of progress in computer hardware, is a matching change in the APIs used to control that hardware. About one year after using our new hardware abstraction layer, the vendor we were working with released a new version of their API. This release dramatically changed the way that telephone call control was handled. Not only did the names of all API functions entirely change, but the associated data structures, state-machines, and usage of this new API were different from before. In effect, it was as though we had switched to a different vendor and had encountered a completely new hardware API. We knew this API would be the only supported interface going forward, and our migration to it was required if we wanted to continue working with this vendor.

The obvious objective was to ensure that our existing application layer code would remain unaffected by this change. If we could meet this goal, we knew it would provide strong evidence that our hardware abstraction was effective.

In order to prevent changes from rippling into the application layer, we knew we could not modify the interface presented by our Call Control Kit. This kit was the direct interface with our application code and was, by design, supposed to be static

regardless of changes to the vendor API. The peer object contained by this kit—a subclass of our `CallControlPeer`—was the proper place to begin.

We rewrote this subclass to use the new API functions provided by our vendor. Some of the new logic required by the change, was hidden inside the peer itself. Other logic however, due to the threading behaviour of our application, could not be isolated in the peer itself. For this logic, we had to expand the generic interface in the base Call Control Peer. We also needed to create a new data structure to house state information for this new logic. This data structure, like all others we had created, followed the Type Laundering pattern—including a new Object Factory to create hardware-specific instances of this structure.

Because this new API required an extension to the Peer interface, and logic changes external to the peer itself, it was necessary to change our Kit class. Fortunately, the double layer of abstraction from the hardware—created by the peer and the kit—was very powerful. We were able to add extra state management, call the new Peer methods, and use the new Peer data structures, without any modification to the external interface for the Kit. While the complexity of the kit increased, the code that called it from the application layer was completely insulated from this change.

With the change complete, we had effectively migrated our call control functions from one API to another without having any affect on application-layer code. This successful transition provided compelling evidence that Type Laundering is a powerful tool for abstracting hardware APIs.

5.3.2 HAL Expansion

The next challenge we faced was a requirement to place our hardware abstraction layer in a system without any installed telephony hardware. This unusual request resulted from a desire to access file translation capabilities built into our hardware abstraction layer, on a system where no hardware would be present.

While the hardware vendor's API supported this type of situation, we had not considered this ability when we designed the hardware abstraction layer. Many of the

peer methods called by our kits required data structures that represented physical hardware in the system.

One possible solution was to create a completely new set of kit and peer functions that were designed specifically for a system without physical telephony hardware. While this would work, we felt it would needlessly expand the API to our HAL. What we preferred was a method that would allow functions to be called the same as before. Those functions that worked without hardware would proceed as expected, while those that could not work without hardware would fail with an exception. To do this we would have to expand our data structures so we could represent the absence of telephony hardware.

Fortunately, these data structures—like everything else in our hardware abstraction layer—made use of the Type Laundering pattern to hide vendor specific details. Similar to the example provided in Section 4.5.4, we had a `Board` base class to represent the idea of a telephony board. This was subclassed as `SomeVendorBoard` to represent physical boards from our chosen vendor at the time. Specific instances of `SomeVendorBoard` were obtained through the `BoardDataFactory`. In order to get the behaviour we desired, we created the `NoBoard` subclass of `Board` to represent a telephony board that was not installed. We then modified the `BoardDataFactory` so it could detect the presence of hardware when a `Board` object was requested. If the board existed, an instance of `SomeVendorBoard` was returned; if hardware could not be found, an instance of `NoBoard` was returned instead. In either case, the caller only knew that they had received an instance of the `Board` class.

The hardware specific peer classes were then modified to differentiate between the two subclasses of `Board`. Those methods that could work without physical hardware, could use either the `SomeVendorBoard` or `NoBoard` structures without concern. Those methods that required physical hardware would have code similar to the following:

Listing 5.1: Recovering Type Information in a Peer Object

```
void SomeVendorPeer::placeCall(string destNumber, Board* inBoard)
{
   SomeVendorBoard* board = dynamic_cast<SomeVendorBoard*>(inBoard);
   if (board == null)
   {
      throw new exception("Physical hardware required");
   }
   // else: found a valid board - place telephony call...
}
```

This neatly allowed us to hide the details of whether hardware was present or not. Application layer code could be written mostly unaware of these changes. The only extra effort needed on the application developer's part was the understanding that some functions in the hardware abstraction layer could now fail if physical hardware was not present, while others could succeed. This complicated the application code slightly, but we could exploit existing exception handling code created for other error conditions, and simply enhance it to deal with these new conditions.

This expansion to our hardware abstraction layer took less than two weeks to design, code, and test, and only a few hours of re-work in the application layer for the extra exception handling. The small impact on the application code was remarkable given that we effectively expanded our hardware abstraction layer to support a new variant of telephony hardware. The developers involved in this project were extremely pleased with the ease with which the hardware abstraction was expanded. This example provided further evidence that Type Laundering is an effective mechanism for designing hardware abstractions.

5.4 Developer Reactions

No formal study of developers' reactions to this pattern has been conducted. However, the author has discussed the experience of using this pattern with several developers, and the results were reasonably consistent. Taken as a whole, developers were very pleased with this pattern, though they did identify a few weaknesses.

Without reservation, developers felt this pattern was very effective at its main goal: abstracting hardware details from application code.

A weakness identified by some developers was that Type Laundering is an unusual idiom; it took more time to learn this pattern and understand what was happening in their code, than other simpler patterns. However, none of the developers felt that Type Laundering was too complex to use regularly.

Another shortcoming identified by developers was the overhead involved in expanding the hardware abstraction interface. If a developer wants to add a new function to a particular *kit*, it is necessary to modify the kit and its interface, the base peer class and its interface, and the hardware specific class(es) and their interface(s). Contrast this with the traditional approach where a single class and its associated interface would be all the changes required. However, all developers agreed that this extra overhead was well worth the benefits provided by Type Laundering.

The last problem mentioned by developers is not actually a weakness in the pattern, but a possible weakness in some development tools. Because Type Laundering makes heavy use of dynamic typing for objects, debugging the application layer often results in references to hardware-specific objects that are typed as their base class counterparts. If the debugger being used is unable to determine the dynamic type of these objects, hardware-specific information will be unavailable in the debugger. Though inconvenient, this does not make debugging impossible. Fortunately, this challenge can be overcome by switching to a more intelligent debugger.

5.5 Comparison With the Bridge Pattern

Readers who are familiar with other Software Design Patterns may wonder about the similarity of Type Laundering to the *Bridge Pattern* (see Figure 5.1).

The purpose of the Bridge pattern is to "decouple an abstraction from its implementation so that the two can vary independently" [8]. This is certainly the same purpose for which the Type Laundering solution was designed. However, there is an important difference in how the Implementation objects are accessed in each pattern.

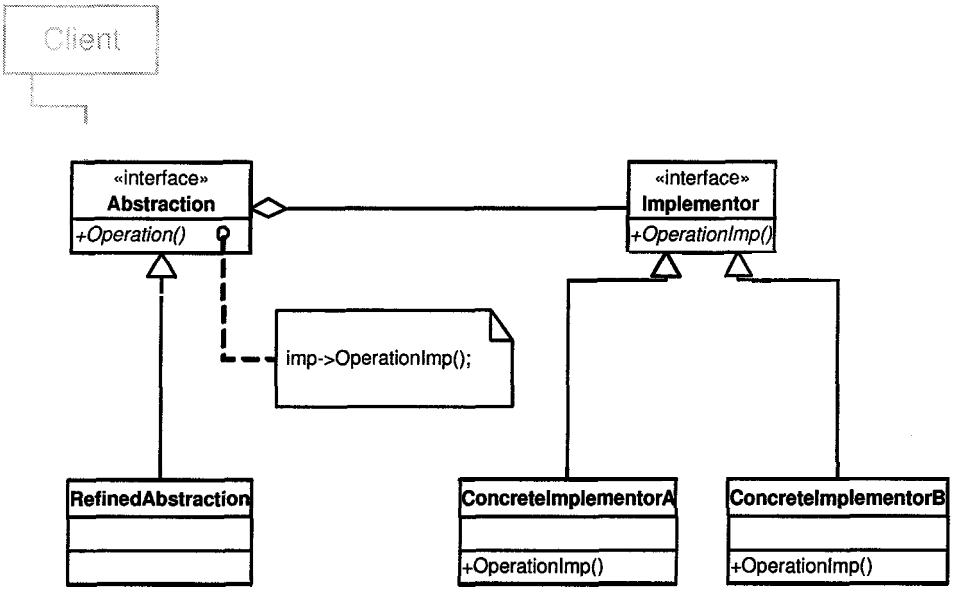

Figure 5.1: Bridge Pattern Structure

In the Bridge pattern, all access to the Implementation object is performed through the Abstraction object. In Type Laundering, the Implementation object is used directly while appearing as if it is the Abstraction object from the code's point of view. In effect, Type Laundering flattens the Bridge pattern to only its Implementation class, while still providing the independent variation features of Bridge.

We believe this difference provides some important advantages over the Bridge pattern. First, there are the issues of performance. Because all calls in the Bridge pattern must pass through the Abstraction object first, Type Laundering can provide more efficient code execution if the Implementation object is called frequently. As well, because an Abstraction is only an interface in Type Laundering, while it is a complete class in the Bridge pattern, Type Laundering can result in shorter build times if you have a large number of classes using the pattern.

More important, however, is what happens when you have a group of cooperating Implementation objects. In this situation, it is reasonable for one Implementor to return to a caller an object that is an instance of a second Implementor class with

the expectation that this object will be passed to a third Implementor or returned to the original Implementor later on. What does this look like?

```
// Control Implementation
class ControlImp
{
  public:
     doSomethingPublic() { //... }

     // This method is unknown to users of our Abstraction
     doSomethingPrivate() { //... }
}

// Control Abstraction
class Control
{
  public:
     doSomething() { controlImp->doSomethingPublic(); }

  private:
     ControlImp* controlImp;
}

// Window Implementation
class WindowImp
{
  public:
     // Return our control Imp so it can be passed
     // to other participants in our hierarchy
     ControlImp* getControl() { return controlImp; }

     // This method uses an internal method from another
     // Implementor.
     someFunction() { controlImp->doSomethingPrivate(); }

  private:
     ControlImp* controlImp;
}
```

```
// Window Abstraction
class Window
{
  public:
    ControlImp* getControl() { return windowImp->getControl();}
    someFunction() { windowImp->someFunction(); }

  private:
    WindowImp* windowImp;
}
```

Because we have a set of cooperating Implementation objects, we need those objects to use each other directly so they can access each other's unpublished interfaces. However, we cannot allow our Implementation objects to return other Implementation objects to callers because they do not know about those types. Instead, our Implementation objects must return Abstraction objects to the callers. This will force our Implementation objects to hold two objects - both an Implementation and its associated Abstraction. It would be nice if we could avoid the need to hold both objects.

Because Type Laundering uses the Implementation objects directly, there is no problem in returning an instance of another Implementation object—you just use Type Laundering on the second type before returning it (see Section 4.5.4 for a description of this principle). Curiously enough, the Bridge pattern could use Type Laundering to clean up its solution to this problem, but that begs the question about whether the Type Laundering pattern should have been used in the first place.

5.6 What About Templates?

Experienced C++ users, or users familiar with other languages that support generic programming, are likely to ask if templates could play a useful role in Type Laundering. Because Templates are a complex subject in their own right, and because there are probably dozens of ways they could aid in the use of our pattern, we cannot provide an exhaustive analysis in this thesis. However, there are a couple of interesting applications that we have considered so far.

5.6.1 TypeLists

A TypeList is a special kind of template construct built around a very simple idea.

```
template <class H, class T>
struct typelist
{
    typedef H head;
    typedef T tail;
};
```

This tiny bit of code can be used to construct, at compile time, recursive types that allow for powerful generic programming techniques. One particularly creative use of TypeLists was proposed by Andrei Alexandrescu in an article for C/C++ Users Journal [9]. In his article, he considers the challenge of adding support for the Visitor pattern [8] to an arbitrary class hierarchy when you do not have access to the original code. The typical solution is presented as follows:

> Consider, for example, a hierarchy rooted in DocumentItem featuring the concrete types TextArea, VectorGraphics, and Bitmap, all directly derived from DocumentItem. To add SomeOperation, you can do the following:

```
void SomeOperation(DocumentItem* p)
{
    if (TextArea* pTextArea = dynamic_cast<TextArea*>(p))
    {
        ... operate on a TextArea object ...
    }
    else if (VectorGraphics* pVectorGraphics =
        dynamic_cast<VectorGraphics*>(p))
    {
        ... operate on a VectorGraphics object ...
    }
    else if (Bitmap* pBitmap = dynamic_cast<Bitmap*>(p))
    {
        ... operate on a Bitmap object ...
    }
    else { throw "Unknown type passed"; }
}
```

He then shows how TypeLists can be used to automate the creation of this code pattern, and even provide compile-type checking when objects are missed. The reader should review Alexandrescu's original article for a complete discussion.

It is not exactly clear how, or even if, this technique can be used to augment or replace our Type Laundering variation. Certainly, this idea does cover some of the same ground as Type Laundering—in fact, Vlissides discusses the Visitor pattern in his original paper. However, it appears that Alexandrescu's methods may have more to offer to Vlissides' original pattern than to our variation, mostly due to the fact that our version has turned the pattern inside-out and hidden the obvious places where TypeLists might be used.

More consideration on this topic might reveal additional applications for TypeLists.

5.6.2 Detecting Convertibility and Inheritance

Another area where templates might offer some benefit to our pattern is centered around the need for dynamic casting base types in the Peer classes (see Section 4.5.2). By relying on dynamic casting, we give away much of the type safety that the C++ compiler can provide, and have to accept that some conversions might fail only at run-time. If a project is deployed with only a single vendor's Peer objects in the code-tree at build time, the problem does not arise. However, that certainly does not describe all applications. It would be nice if we could determine at compile-time if a given `dynamic_cast` was valid or not.

Templates may have a way to make this check possible. The idea centers around the hidden power of the C++ `sizeof` operator. Under C++, the `sizeof` operator can return the size of any C++ expression, no matter how complicated. This means that `sizeof` understands overloading, template instantiations, conversions rules, and everything that can form part of a C++ expression.

In another article, Andrei Alexandrescu shows how to use the power of `sizeof` to create a series of types that can be used at compile time to determine if two types

are convertible [10]. If two types support the correct kind of conversion, you can do away with the `dynamic_cast` completely and gain not only compile-time checking, but also improved efficiency.

So how do these type-conversion checks work? Perhaps it is best if we let Alexandrescu explain this in his own words:

> "The idea of conversion detection relies on using `sizeof` with overloaded functions. You cunningly provide two overloads of a function: one accepts the type to convert to `(U)`, and the other accepts just about anything else. You call the overloaded function with `T`, whose convertibility to `U` you want to determine. If the function that accepts a `U` gets called, you know that `T` is convertible to `U`; if the 'fallback' function gets called, then `T` is not convertible to `U`.
>
> To detect which function gets called, you arrange the two overloads to return types of different sizes, and discriminate with `sizeof`. The types themselves do not matter, as long as they have different sizes."

This is done by creating two types of different sizes.

```
typedef char Small;
struct Big { char dummy[2]; };
```

Next, you create two overloads.

```
Small Test(U);
Big Test(...);
```

Because of the type-conversion rules in C++, the ellipsis operator is always the last match performed by the compiler, and is guaranteed to work as our fallback function. Given this set of definitions, it is easy to create a simple bit of code that can tell you if a conversion exists between two types.

```
const bool convExists = sizeof(Test(T())) == sizeof(Small);
```

Finally, we package this whole thing up into a single template structure:

```
template <class T, class U>
class Conversion
{
    typedef char Small;
    struct Big { char dummy[2]; };
    Small Test(U);
    Big Test(...);
    T MakeT();
public:
    enum { exists = sizeof(Test(MakeT())) == sizeof(Small) };
    enum { exists2Way = exists && Conversion<U, T>::exists };
    enum { sameType = false };
};
```

This structure provides three enums that evaluate to different values depending upon the convertibility of the types in question. When two types are exactly the same, all three of these enums should evaluate to true. For this reason, we implement the `sameType` specialization of `Conversion` that takes the same type twice: it will be automatically selected by the compiler if both template types are identical. This specialization returns the proper value for each enum when both types are the same.

```
template <class T>
class Conversion<T, T>
{
public:
    enum { exists = true, exists2Way = true, sameType = true };
};
```

These two definitions can now be used at compile time to determine if two types have an inheritance relationship.

```
#define SUPERSUBCLASS(B, D) \
    (Conversion<const D*, const B*>::exists && \
    !Conversion<const B*, const void*>::sameType)
```

The second portion of this conjugation ensures that we are not trying to check a type's convertibility to `void*`. Under the rules of C++, all types can be converted to `void*`, so we specifically disallow this check if `void*` is the superclass being used.

Admittedly, this is all very confusing, especially if you are not comfortable reading template code. But it does provide C++ programmers with the ability to check inheritance at compile-time.

The big question is whether any of this is useful to our Type Laundering variation. At first glance, despite the cleverness of the technique, it does not appear to offer much help. After all, we already know that the types we are trying to test all inherit from a single base class, the appropriate `Peer`, so a straightforward conversion check will always succeed.

Perhaps there is a way to enhance our solution so this technique can be employed. Ideally, we could use this technique to provide for template-driven creation of methods that were only applicable to the type of the object being passed. This would ensure that incorrect types would not have a matching Peer function and the compile would fail. We have not looked too far into this concept but feel certain that there is a solution here waiting to be discovered.

5.7 Final Assessment

In Section 3.3 a number of shortcomings in the traditional approach to hardware abstraction were highlighted: hard-coded vendor classes, problems determining object types, the use of vendor specific methods, and poor information hiding. A Type Laundering based hardware abstraction layer removes all four of these weaknesses. Let us consider how Type Laundering addresses each of these points in turn.

Hard-Coded Vendor Classes These still exist in the Type Laundering solution, but they are completely hidden from application layer code. Changes to these classes, or even their complete replacement, will not affect code outside the hardware abstraction layer.

Determining Object Types This determination is handled by one or more factories in the Type Laundering solution. While code for hardware determination must

still be written and maintained, it is completely isolated inside the factory. Application layer code does not need to know anything about installed hardware.

Vendor Specific Methods These are accessed only through hardware-specific peer objects. The peer objects are accessed through associated kit classes. This design separates application-layer code from vendor specific methods by at least two layers of abstraction. Changes to vendor specific APIs can be completely contained within the peer and kit layers, ensuring the application layer is unaffected.

Poor Information Hiding Exposure of vendor-specific hardware functions and structures is not necessary in a Type Laundering solution. The traditional solution requires application code to access vendor files and data structures directly. A Type Laundering design uses object factories and abstract classes to hide vendor specific data structures, and hardware abstraction internals, from application layer code.

6. Contributions

This thesis provides additional understanding about Type Laundering as a tool for software abstraction. In particular, it shows a way in which Type Laundering can function as a hardware abstraction mechanism, and highlights additional support that is required for the pattern when used in this manner.

6.1 Type Laundering in Reverse

We have shown that the Type Laundering software pattern has application beyond Vlissides' original proposal as a mechanism for adding capabilities to a framework library. By using the Type Laundering pattern in reverse, we have demonstrated how it can become a powerful tool for hiding functions and data types from higher-layer code. We have explained how this pattern reversal is accomplished and demonstrated that it solves many of the weaknesses inherent in the traditional C++ solution to hardware abstraction.

We have also shown how reversed Type Laundering creates a new challenge related to the construction of laundered types and the use of the Factory pattern: a challenge which is not present when Type Laundering is used according to Vlissides' original proposal. We provided some understanding of this challenge, showed our particular solution, and proposed ways in which our solution could be improved.

6.2 Kits Are Needed

When the idea of a Kit layer was originally suggested, we expected it to be useful, but there was some question about whether it was really necessary. Now that we are able to look back on this pattern with a few years experience, we can say with confidence that the Kit objects are a fundamental part of this design pattern. While Type Laundering is a useful concept on its own, the secondary layer of indirection provided by the Kit is essential to properly isolate the logic of each vendor's hardware

from the application layer.

In effect, the Type Laundering pattern provides abstractions for the hardware's API and Data structures, while the Kit provides abstractions for the hardware's logic. This has led us to believe that the following principle is generally true across many (if not all) forms of hardware abstraction: namely, that at least two layers of indirection are always required. The first layer hides structure (API and Data Types), while the second layer hides logic (data validation and state manipulation). It is our belief that attempting to perform a hardware abstraction where both of these responsibilities are combined into a single layer of indirection is likely to present great challenges as the software evolves.

6.3 A Better Bridge

In our evaluation we compared Type Laundering to the Bridge pattern (see Section 5.5). We showed how Type Laundering achieves the same benefits as Bridge, while adding a number of improvements for program performance and for the passing of implementation-specific data types. It is our opinion that Type Laundering should be used in place of the Bridge pattern in all instances where an adequate solution to the factory problem can be found (see Section 4.6). While there are still situations where the simpler structure offered by Bridge is adequate, any sufficiently complex abstraction layer will find Type Laundering a more powerful tool and should consider its use in place of Bridge.

7. Conclusion

This section describes some avenues for future investigation into the use of Type Laundering.

7.1 Future Work

While our Type Laundering solution has been very effective in practice (see Section 5.1), there are a few areas that would benefit from closer examination.

Memory Management This can be especially challenging with this pattern. Because object factories return objects as pointers (or references), it is very important to ensure proper deletion of these objects. Many authors have considered solutions to this problem, including Vlissides himself in his original Type Laundering paper [7], and there is active debate surrounding this issue. Of course, if the implementation language offers garbage collection, this is much less of an issue.

TypeLists The TypeList technique discussed in Section 5.6.1 may have possible applications to the Type Laundering pattern. TypeLists offer an alternative to certain forms of the Visitor pattern, and those same forms may provide ways to leverage template techniques in Type Laundering.

Dynamic Casting Runtime type identification is an essential part of a Type Laundering design. Fortunately all modern object-oriented programming languages support this functionality. However, Type Laundering cannot be used in a language where this feature is absent. However, it may be possible to remove the need for runtime type casting by using templates and compile-time code generation (see Section 5.6.2). When a language is used that provides support for runtime reflection (such as Java), the benefits of Type Laundering may have to be re-evaluated.

Object Factory Design The ideal design for a Type Laundering factory is open to debate. A hardware abstraction layer can be built using a single object factory capable of laundering all exposed types or a series of smaller factories focused on specific groups of objects. Which method should be used is probably best determined by the needs of each individual application, though there may be additional patterns that could be employed in this area.

Object Factory Factories The idea here is to use a single meta-factory to create all other factories. These secondary factories would in turn create the individual Type Laundered objects. Investigation into this concept may provide interesting solutions for the problem of modifying factory output whenever support for new hardware variations are changed at run-time.

7.2 Summary

This thesis has outlined the problems inherent in the traditional approach to hardware abstraction. It has shown how a Type Laundering based hardware abstraction layer can be constructed, provided practical examples of this pattern in use, and has shown that Type Laundering is a powerful pattern for resolving all the key weaknesses associated with the traditional approach to hardware abstraction.

Bibliography

[1] Lionel C. Briand, Premkumar T. Devanbu, and Walcelio L. Melo. An investigation into coupling measures for C++. In *International Conference on Software Engineering*, pages 412–421, 1997.

[2] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, January/February 1999.

[3] Brad Appleton. *Patterns and Software: Essential Concepts and Terminology.* Available at `http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html`.

[4] NMS Communications Inc. *AG 4000 Installation and Developer's Manual*, 2003. Available at `http://www.nmscommunications.com/manuals/60003-18/default.htm`.

[5] NMS Communications Inc. *CG 6500C Installation and Developer's Manual*, 2003. Available at `http://www.nmscommunications.com/manuals/62022-13/default.htm`.

[6] John Vlissides. Type Laundering. *C++ Report*, Feb, 1997.

[7] John Vlissides. *Pattern Hatching, Design Patterns Applied.* Addison-Wesley, 1998.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[9] Andrei Alexandrescu. Generic programming: Typelists and applications. *C/C++ Users Journal*, Feb, 2002. Available at `http://www.cuj.com/documents/s=7986/cujcexp2002alexandr/alexandr.htm`.

[10] Andrei Alexandrescu. Generic programming: Mappings between types and values. *C/C++ Users Journal*, Oct, 2000. Available at `http://www.cuj.com/documents/s=8002/cujcexp1810alexandr/alexandr.htm`.