Project Number. GFPMS08

Microsoft MQP

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

_____

Jared Renzullo

Tyler Boone

Date: 24 April 2008

Approved:

_____

Professor Gary F. Pollice, Major Advisor

1. Microsoft
2. API
3. Power Management

Abstract

The MQP presented in this document was completed at Microsoft in Redmond, WA between January and March 2008. The project was to design, implement, and test a power management application programming interface (API) for use in the .NET framework. The API interfaces with existing Windows APIs for power management, some of which are new to Microsoft Windows Vista, to provide a limited subset of functionality to .NET developers.

Acknowledgements

We'd like to thank Microsoft for providing us with the opportunity and resources to complete this project. We'd also like to thank our mentor, Melitta Andersen and the BCL team, namely Matt Ellis and Robert Villahermosa.

Table of Contents

List of Tables

1. Introduction

The .NET Framework is a programming framework which supports a variety of languages including: C++, C#, Visual BASIC and Python; which are all currently among the top 10 most popular programming languages (1). A programming framework is a system that application developers can utilize to make writing applications easier. Frameworks differ from libraries in that libraries are used along with code to accomplish a task, whereas applications can be said to be "built upon" a framework. A framework thus provides a layer of abstraction to program developers as opposed to simply a set of functionality. Managed code refers to all code written on the .NET framework which is compiled to intermediate language (IL) and executed by the Common Language Runtime (CLR). Unmanaged code refers to any code which is compiled directly to machine code. In this MQP, we use the term "unmanaged code" to refer to C or C++ code compiled into Windows native executables. The .NET framework provides the ability for managed code to interact with unmanaged code. The framework also encapsulates difficult operations in a Base Class Library (BCL). The BCL simplifies common programming functions such as file reading and writing, XML parsing, etc.

Microsoft introduced several new APIs as part of the Windows Vista operating system, one of which is an updated power management API. The power management API allows code to query the system's current power state (i.e. battery, AC, etc), register for power notifications (i.e. when the power source changes, when the battery life changes, etc), handle shutdown notifications, and more. For this MQP, we designed, implemented, tested, and deployed an API for managed code to easily access the unmanaged API. Currently in order to access Vista's power management API, .NET programmers have to use platform invoke (p/invoke) to directly access the function calls. P/invoke requires the programmer to have knowledge of the .NET data

marshaling system as well as Win32 data types and other advanced concepts. In an effort to simplify access to Vista's new power management features, Microsoft decided a managed API should be developed which would encapsulate all aforementioned advanced concepts. We worked as part of the BCL team to design this managed API. Normal project teams in the CLR group are composed of a dedicated program manager, one or two developers and one or two testers; however, we both acted as program manager, developer, and tester for the project.

The project began with the managed API's initial design. The original intent was to utilize the majority of the Win32 power management library's functionality. After discussions with members of the power management team in the Windows group, it was decided that much of the original API's functionality does not need to be exposed in a managed library. The managed APIs are a means to make certain features easy to use for application developers, and some of the low-level operations are more in the realm of systems developers. The new API supports registering for various power notifications such as current power source changes and battery life changes as well as querying for values directly.

The newly revised design was sent to a program manager on the BCL team for review. By following his suggestions, the API was condensed into a centralized class which made the usage scenarios simpler. After incorporating these changes, we conducted a design review where all members of the BCL team had a chance to comment on the design. Present at this review were developers, testers, technical writers, and program managers. Minimal changes were made to the design as a result of the review.

After completing the design, the next step was to actually implement the API as well as develop a test plan and test cases. The test plan contains an overview of the feature being tested

as well as a description of how each component of the feature will be tested. Any specific notes or boundary conditions to consider during testing are included in the spec.

The remainder of this document includes a detailed look at the API's design, implementation, and testing stages. Chapter 2, the background section, includes a more detailed look at the .NET framework and the advanced Windows programming concepts that were encapsulated as a result of the project. A brief history of Microsoft is also presented. In chapter 3, the methodology section, details on what software engineering concepts and design patterns were utilized in the design and development phases are included. A detailed look at the new managed API is given. Code samples demonstrate usage of the new API. An in depth look at the design decisions made during the design phase is also given. The methodology section also contains information about the testing portion of the project and how certain hurdles specific to testing a power management API were dealt with. Chapter 4, the results and analysis section, provides a look at the quality of the code written for the API. Code metrics are discussed as well as reactions from the BCL team on the final product. Chapter 5, the conclusions section, overviews the relative successes and failures of the project as well as the lessons learned.

2. Background

Microsoft Corporation began as a partnership between two computer enthusiasts, Bill Gates and Paul Allen in 1975. The company grew steadily during the 1970's, shipping versions of Microsoft BASIC on multiple platforms. In the early 1980's, Microsoft developed the Microsoft Disk Operating System (MS-DOS) which ran on the IBM personal computer as well as a multitude of other platforms (2). Microsoft also developed software such as a word processor,

spreadsheet application, and flight simulator. In 1985, Microsoft released the first version of its operating system called Windows (2). In the 1990's, eight versions of Windows were released, starting with Windows 3.0 and ending with the release of Windows 98 Second Edition (3). Today, the company generates over fifty billion dollars in revenue per year and is a clear leader in software (4).

A dynamic-link library (DLL) is a Microsoft file type, meant to save both memory and disk space. Code common to multiple programs is compiled to a DLL, loaded into memory, and then used by multiple different processes. Microsoft Windows includes many different DLLs which can be accessed by programs in order to interact with Windows' features. The power management API includes functions in several different DLLs such as PowrProf.dll, Kernel32.dll, and User32.dll. For a C++ application to access Vista's power management features, the programmer includes the proper header files and links with the proper library files (.lib). The necessary DLLs are accessed at runtime.

The managed API we created was written entirely in C#, a language created by Microsoft in 2001 for the .NET framework. C# is an object-oriented programming language, similar to both Java and C++. Like Java, which runs on its virtual machine, C# runs on the .NET CLR. C# can call functions written in unmanaged C++ using platform invoke (p/invoke). Essentially an unmanaged DLL is imported and a C# signature is written which targets a function in the DLL. The C# programmer needs to be aware of data marshaling, moving data from one language to another. Windows programs written in C++ do not use the exact same data types as C#. At a very basic level, the programmer enters the C# data types in the p/invoke signature which correspond to the data types in the unmanaged function signature. For example, if the unmanaged function has two parameters, a HANDLE and a SHORT, the managed signature will

have an IntPtr and an Int16. Sometimes it is also necessary to explicitly tell the CLR how to marshal some data types. For example, when filling a managed struct with data from an unmanaged struct, if one of the members of the unmanaged struct is an eight bit boolean and the managed version is an thirty-two bit bool, the data needs to be explicitly marshaled as one byte. If the data is not explicitly marshaled, the managed bool will contain one byte from the proper member as well as three bytes from the struct's next member. Figure 1 shows an example p/invoke signature that includes marshaling of parameters and the return type.

```
[DllImport("kernel32.dll", SetLastError = true)]
[return: MarshalAs(UnmanagedType.Bool)]
internal static extern bool SetConsoleCtrlHandler(
    ConsoleCtrlDelegate HandlerRoutine,
    [MarshalAs(UnmanagedType.Bool)] bool Add);
```

Figure 1

Windows Vista includes new power management functionality through its API for developers. Programs are able to query the system for current power information i.e. the current power source, current battery life, etc. In addition to querying current power information, programs can register to receive notifications of six different power events. The first event that can be registered is a power personality change event. A power personality can be one of three values: High Performance, Power Saver, and Automatic. In the Windows control panel for power management, users select a power plan to use. A power plan contains a set of values for all power management settings. Every power plan (including user created ones) is assigned a power personality, which identifies the power plan's intent to programs. By registering for a power personality change event, programs can modify their behavior to reflect the new power personality. For example, if the power personality is changed from High Performance to Power

Saver, a program might stop performing power intensive background tasks and use only necessary functionality.

Another power event that programs can receive notifications for is a power source change event. A power source can be AC power, a battery, or an Uninterruptable Power Source (UPS). Programs should be aware when the power source changes for several reasons. If the power source changes to a UPS, programs should immediately save all unsaved work and prepare for the system to shutdown at any time. A UPS is very short term and is only meant to keep the system on long enough after a power failure to be shut down properly. If relevant, programs should also modify behavior when the power source changes from AC to battery. Programs should do all they can to maximize battery life. In addition to a power source change, programs can be notified when the battery life changes. How often the notification is sent is dependent on the individual system, but battery life is always rounded to a whole percent. This notification can be used, for instance, to prepare for unexpected shutdown when the battery life is low.

Programs can register for a system busy notification. This notification is sent when the system is not likely to enter an idle state in the near future. It is mainly useful to programs wanting to perform background tasks. If the system is already busy, it's safe for background tasks to be run since there is no danger of preventing the system from idling. Windows can also send a notification when the system enters or exits away mode. This notification, however, is not supported by our managed API as it was determined to be an infrequently used feature.

The last notification programs can register for is a change in monitor status. A notification is sent whenever the system turns the monitor off because of user inactivity. It is also sent when the monitor is turned back on. When the monitor is turned off, applications with graphics should

stop rendering content to the screen in order to save power. When the monitor is turned back on, rendering can continue. (5)

In Windows Vista, Microsoft introduced new shutdown blocking capabilities for programs. Previously in Windows XP, programs were notified when a shutdown was occurring so they could display a warning to the user for whatever reason, i.e. a notepad document was modified but not saved or a CD was in the middle of burning. In Windows Vista, the process is more streamlined. A distinct UI prompt is given to the user during a system shutdown. The UI shows all currently open programs, each with an optional reason for why the program needs to remain open. This can be viewed inFigure 2: Shutdown Blocking Screen. From this screen, shutdown is either canceled or confirmed at which point all programs have five seconds before they are automatically closed. Through the Windows API, programs are able to block shutdown proactively and set a reason, which is displayed in the UI prompt. (6)

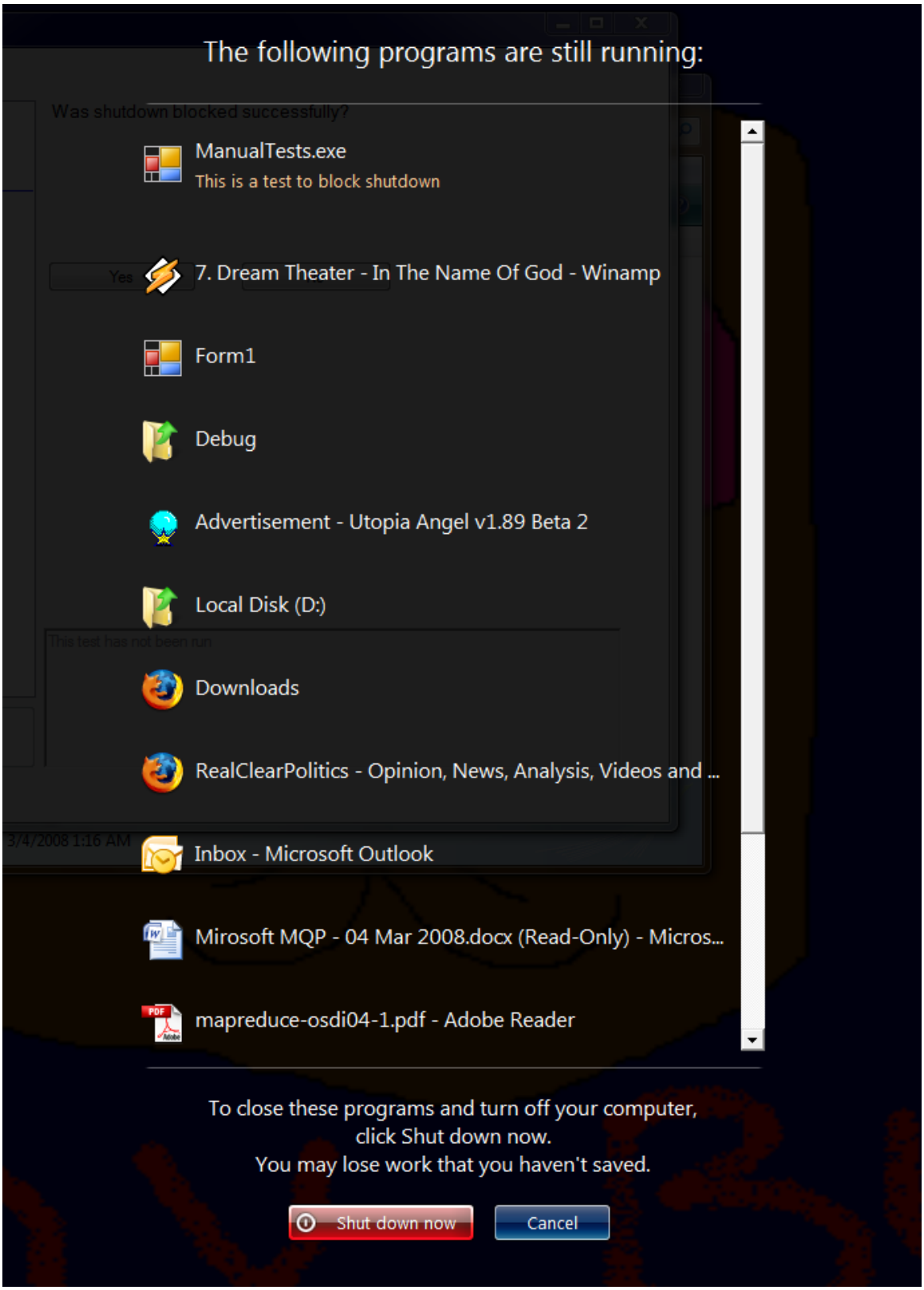

Figure 2: Shutdown Blocking Screen

3. Methodology

3.1 Design

Original Design

The original design was created in an effort to include (just about) everything in the Win32 power management API. We took great pains while creating this design to follow all the design guidelines presented in the Framework Design Guidelines book (7). Following these rules, we hardly had any properties (34) as compared to methods (188). While the book suggests using Properties instead of getters and setters, it says not to use a property when the operation can take longer than direct memory access. This design is massive and it is fortunate that we scaled it down. Here are some metrics for the original design:

- Public methods: 188
- Public properties: 34
- Public fields: 0
- Public events: 6
- Public types: 24
    - 19 classes
    - 1 interface
    - 4 enumerations

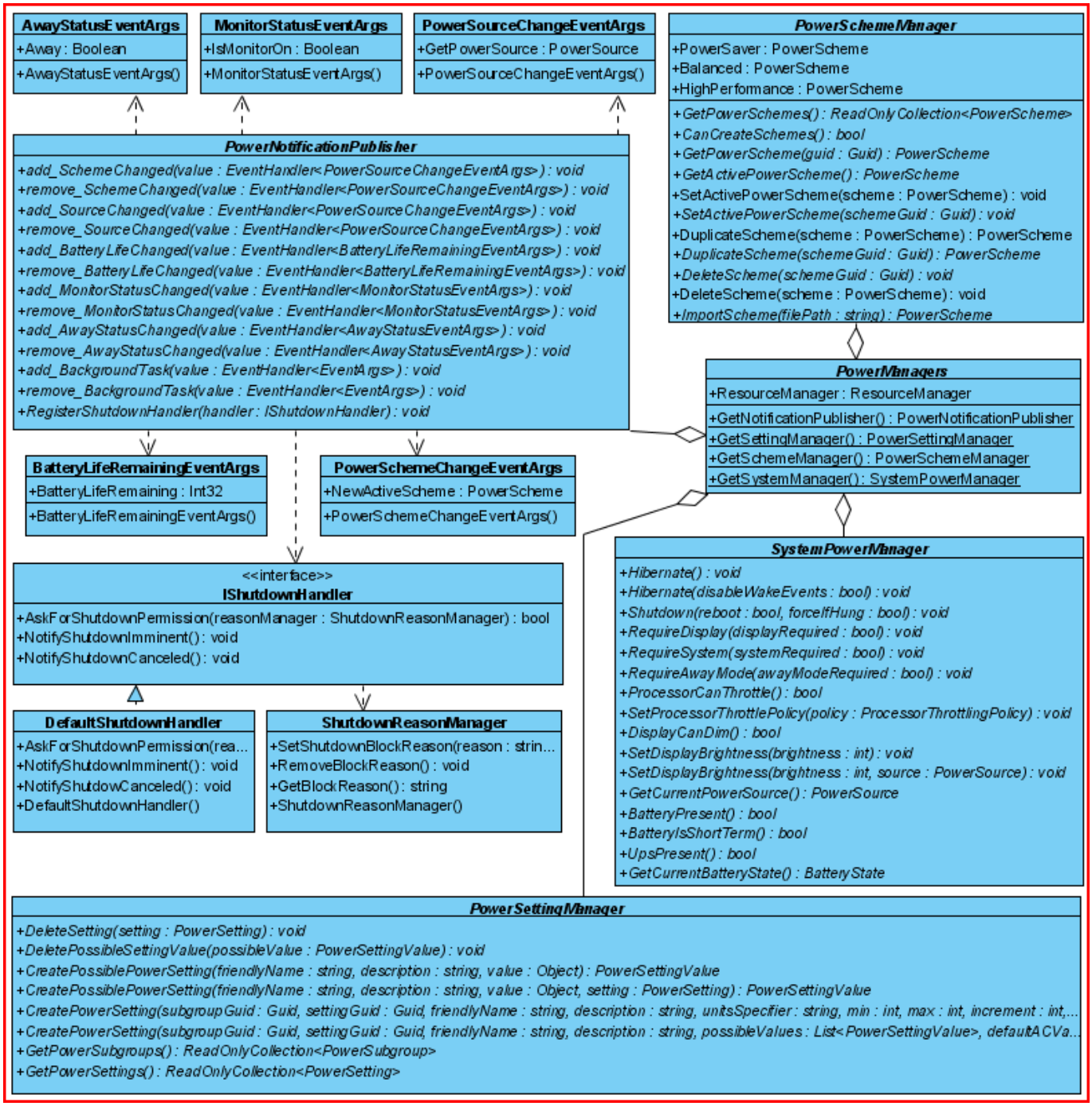

Figure 3: Original API

Figure 3 comprises the managers and the custom EventArgs. In an effort to compartmentalize the various functionalities the managers were broken into 4 submanagers. All of them were going to be singletons available from the PowerManagers static class. Most of the events fire custom EventArgs that contain information about the event. For instance, the BatteryLifeRemainingEventArgs contains

the percentage of battery life remaining. The PowerSettingManager is used to create and modify power settings as well as list the setting subgroups. The PowerSchemeManager is used to create and modify power schemes.

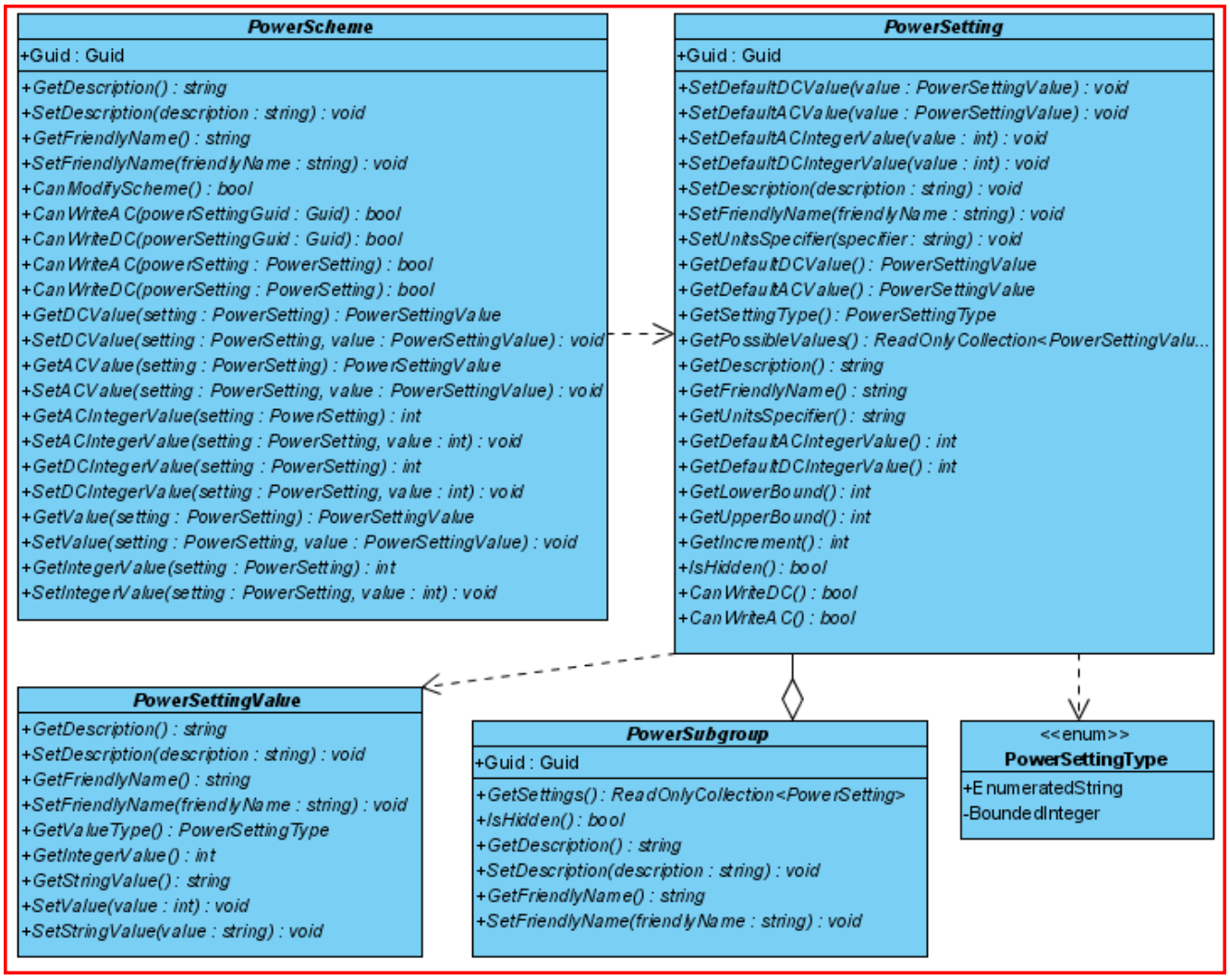

Figure 4: Power Schemes & Power Settings

Figure 4 shows the structure of the power settings and power schemes. A Power scheme is a set of values for each power setting for both AC power and DC power. There are two types of power settings as noted by the PowerSettingType. EnumeratedStrings are power settings where the user selects one of a given list of choices. BoundedIntegers are settings that have integral values and can have a minimum and a maximum.

Revised Design

The revised design was created after some feedback from the power management team in the Windows group. They were concerned about the managed API providing too much low-level functionality. With their help, we determined some of the primary scenarios for the managed application developer. The revised design focused on those instead of trying to expose all of the intricacies of the native API.

Final Design

After receiving feedback on our design from BCL program manager Justin van Patten, we redesigned our API. Following a design review, the API's design was finalized.

Design Decisions

The native Windows API allows programmers to block shutdown before one has been initiated. When designing the shutdown blocking portion of the API, we considered several possibilities. One possible design was to only allow for reactive shutdown blocking. Users could register for two events using the API, SessionEnding and SessionEnded. The SessionEnding event is triggered whenever a user initiates a system shut down or log off. The SessionEnded event is triggered when the system is actually shutting down. In order to block shutdown, a programmer would access the SessionEnding's event argument in their event handler. The argument would be an object which could be used to block shutdown and set the reason for blocking. The problem with this approach is that it does not allow for proactive shutdown blocking unlike the Vista API.

An alternative design includes both the SessionEnding and SessionEnded events, but instead of passing an object as an event argument to SessionEnding in order to block shutdown, a separate BlockShutdown method is used. With this design, shutdown can be blocked proactively

and reactively. The BlockShutdown method also handles blocking shutdown in multiple places in a single process. Shutdown can be blocked when a CD is being burned, for instance, with a reason of "CD is burning." Shutdown can also be blocked when a setting has been modified, but not yet saved with a reason of "Unsaved settings." In this example, if a shutdown is initiated, the Windows UI would show the process with the reason set to, "CD is burning." and on a separate line, "Unsaved settings."

UML (API only)

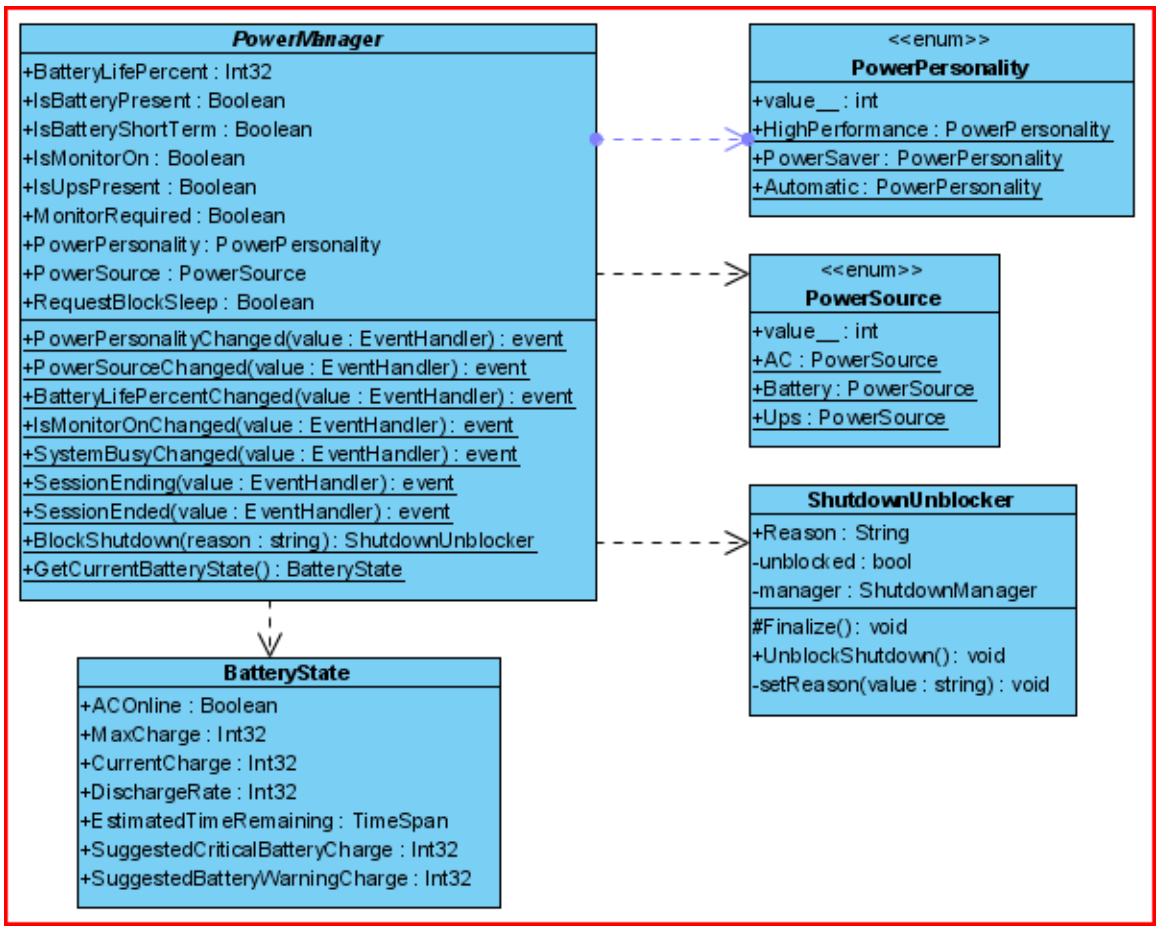

Figure 5: Final API

Figure 5 contains the UML for the final API we designed. The PowerManager class is a static class which is used to register for events, block shutdown, and query or set certain power related

values. ShutdownUnblocker is a class which is returned after blocking shutdown. The class can be used to change the reason for blocking and also to unblock shutdown.

Usage Scenarios

- Register/unregister a function to be called when…
    - The power personality changes
    - The system power source changes
    - The battery capacity changes
    - The system is busy
    - The monitor turns on or off
- **Battery**
    - Determine what power source the system is using (i.e. battery)
    - Check if the battery is currently charging or discharging
    - Get the approximate amount of time the system has left given current power usage.
- Tell the system to keep the monitor on indefinitely
- Set the system to never idle
- Block the computer from shutting down, while also providing a reason to the user for the blockage
- Determine if a battery and/or UPS is present on the system.

Figure 6 is a code sample for registering for a power source change event and determining what the current power source is.

```
PowerManager.PowerSourceChanged += HandlePowerSourceChange;

public static void HandlePowerSourceChange(object sender,
EventArgs args)
{
     PowerSource powerSrc = PowerManager.PowerSource;
     switch (powerSrc)
     {
          case PowerSource.AC:
               Console.WriteLine("AC Power now active.");
               break;
          case PowerSource.Battery:
               Console.WriteLine("Battery now in use.");
               break;
          case PowerSource.Ups:
               Console.WriteLine("UPS now in use.");
               break;
     }
}
```

Figure 6

Figure 7 is a code sample which shows how to tell Windows to never shut down the monitor (i.e. for a video player).

```
PowerManager.MonitorRequired = true;
```

Figure 7

Figure 8 is a code sample for determining if the monitor is currently on.

```
if (PowerManager.IsMonitorOn)
{
     Console.WriteLine("The monitor is on, begin showing
video.");
}
```

Figure 8

Figure 9 is another code sample showing how to block system shutdown.

```
public static void HandleShutdown()
{
    //application startup
    MyApp app = new MyApp();
    while (true)
    {
        System.Threading.Thread.Sleep(3000);
        using (ShutdownUnblocker unblocker =
PowerManager.BlockShutdown("saving"))
        {
            app.save();
        }
    }
}
```

Figure 9

3.2 Implementation

Details

The unmanaged Windows functions used by our API are accessed via p/invoke in a static class called NativeMethods. It is Microsoft convention to include p/invoke calls in such a class. In addition to the Win32 specified function signatures, the NativeMethods file also contains all the structures needed by the functions. A thin wrapper of the native methods is provided in a static class called Power. This class handles all low level Win32 behavior, such as output parameters, memory allocation, etc. All code in our API which needs to access any unmanaged power management functions makes calls directly to the Power class, never to NativeMethods.

The power notification events for power personality change, power source change, system busy, battery life change, and monitor status all use the native method RegisterPowerSettingNotification. The problem with this method is that it requires a window handle as one of its parameters. Whenever the event being registered occurs, Windows sends a message to the window with the specified handle. In order to make the

API flexible enough to be used by multiple types of applications, a hidden form is used to handle all notifications. When an event is registered with the API in the PowerManager class, the MessageManager class is called. The MessageManager class stores all event handlers once they are registered to a particular event. The class also contains the hidden form as one of its members. RegisterPowerSettingNotification is then called with the handle from the hidden form as well as the id of the event as parameters. Windows sends messages to forms through their WndProc method. When a power notification (Windows message) is received, if the notification is of an updated value, the PowerManager class updates its internal cache of the value and then calls all registered event handlers for the notification.

The shutdown blocking mechanism uses an internal ShutdownManager class. This class acts as a mediator to the various ShutdownUnblockers that are active in the system. The shutdown manager uses p/invoke to control the reason for blocking shutdown. When a ShutdownUnblocker is created or its reason is changed, it notifies the ShutdownManager so that it can change the reason. The ShutdownManager also handles the SessionEnding and SessionEnded events. When an application calls the BlockShutdown method, it gets back a ShutdownUnblocker. It must keep track of this object, because if there are no references to it the garbage collector will finalize it. The ShutdownUnblocker has a destructor that will cause it to unblock shutdown when it is garbage collected. This might cause inconsistent behavior in poorly written code; however, we felt it prudent to err on the side of caution and unblock the shutdown. If this was not the case, then a program that did not do anything with the ShutdownUnblocker would block shutdown for as long as the application was running. One implication of relying on the garbage collector to finalize the ShutdownUnblockers is that the ShutdownManager could not keep a reference to any of the unblockers. To get

around this, each unblocker is assigned a unique key when it is created, and the ShutdownManager caches the reason string for the unblockers based on this key. ShutdownUnblocker also implements IDisposable, which allows application to use them within “using” statements which will automatically unblock shutdown when the scope of the “using” statement ends.

Because PowerManager caches values, they need to be updated every time the actual values change. For all properties that are updated by events (PowerSource, BatteryLife, IsMonitorOn, and PowerPersonality), the first time they are accessed, a dummy event handler is registered for the corresponding event. This ensures the cached value is updated every time the actual value changes since the WndProc method will update the value when the corresponding event is fired.

The MonitorRequired and RequestBlockSleep methods both use the native method SetThreadExecutionState. MonitorRequired tells Windows to not shut the monitor off when the system goes idle. A problem we discovered with this method late in the project is that it does not disable the screen saver. We were going to try to make MonitorRequired also disable the screen saver, but there was no appropriate way in the Windows API to disable it on a process by process basis. The screen saver could be disabled globally, but this is not safe in the case that the program crashes or the process is killed by the user. Normally a program that wants to block the screensaver will catch the SC_SCREENSAVE message from windows and consume it without sending it to the DefWndProc method. However, a window or form only receives this message if it is a top level window. Since our hidden form is not top level, it does not receive it, and cannot block the screen saver. Applications that want to do this will need to override the WndProc method themselves and block the screen saver.

All other properties and methods had a more straight forward implementation. Properties not updated by an event are not cached since there is no way to keep the values up to date. Instead, every call to the property calls the corresponding unmanaged function. The BatteryState class contains a snapshot of current battery info. In the class' constructor, data from a call to the unmanaged function CallNtPowerInformation is marshaled into the class.

Design Patterns

The PowerManager class in our API provides an easy to use interface to the end user which is completely different than the unmanaged interface we are trying to wrap, an example of the façade pattern. The PowerManager class is a façade to the Power class. All of the properties in the PowerManager class use lazy initialization to increase performance. The value used by a property is not set until the first time code "gets" that property.

3.3 Testing

Testing Focuses

As part of a comprehensive testing strategy it is necessary to consider much more than simple code coverage and corner case coverage. While correctness testing is very important, professional libraries should be thoroughly tested for security vulnerabilities, proper member visibility, and proper handling of invalid arguments and operations. As part of our comprehensive testing strategy, we have tests for all the following testing focuses.

1. Visibility Testing

Visibility testing is a test that checks for the proper visibility of all methods, fields, events and properties. This testing is important to ensure all functionality exposed in the spec is made public so it can be used by application developers. Furthermore, it is important to make sure there are no members or classes in the namespace that are public and not specified in the spec.

The positive testing is implicit within other testing (if the IsMonitorOn property was private, the test for that property would not compile), however, the negative testing requires the use of reflection. Using reflection, we obtain a list of all the classes and ensure that only the API visible classes are public. We then list all the public members of these classes and ensure there are no public members not specified in the spec.

2. Correctness Testing

This is the kind of testing that is often thought of as primary testing. This testing should cover all user scenarios and identified corner cases. This testing focuses on ensuring that, given valid arguments, state, and environment, the API functions as expected.

3. Negative Testing

Negative testing covers scenarios mostly dealing with catching semantic errors in application code. The purpose of negative testing is to raise a friendly error message to signal invalid conditions or arguments. This helps programmers develop applications by making semantic errors cause the system to fail earlier. This also avoids causing cryptic error messages and invalid data. One example of invalid data is if an attempt is made to check the battery life percentage when no battery is attached. While some implementations might return trash values (0 for instance), it is preferable to throw an exception to alert application developers that the program is likely doing something wrong.

4. Platform Testing

Platform testing ensures that the code is allowed to run when the operating system is a high enough version, and that a PlatformException is thrown when the code is run on an operating system that does not meet the minimum requirements. The spec lists the platform required for all the methods and properties to be used. The minimum requirements range from Windows 2000 to Vista for some of the newer native APIs. This testing is required because all of our

functionality is backed by a Win32 API that we call with p/invoke. Since the .NET assemblies are not compiled with the Win32 SDK header files and are compiled once for all platforms, we cannot rely on compile time directive and compile time and link time checks. P/invoke uses dynamic library loading, and because of this, if we did not do platform checking, the p/invoke attempts would generate unfriendly exceptions that would be hard for an application developer to debug and diagnose.

Test Design

Framework

In our testing we used the Visual Studio built-in unit testing framework (8). This framework is extremely similar to JUnit 4.0 (9). Each test is a class, which can have any number of test methods. The class is instantiated one time and then all tests in that class are run. You can have setup and teardown methods for the entire class, and setup and teardown methods for the individual functions. All of these methods are identified by annotations as they are in JUnit 4.0.

Tools

During the development process we used Visual Studio's built in test runner to run the unit tests. However, the BCL testing team does not use these kinds of tests. Their testing tool is a simple online system. A test consists of an optional setup program, an optional cleanup program, and the test program. The test program is simply an executable file that the online tool can be configured to pass command line arguments to, if necessary. The test is marked as a pass if the test program returns with exit code 0, otherwise it is marked as a fail.

To use this tool, we developed a command line tool that takes the name of a test as a command line argument and runs it. Furthermore, it can take the name of a class and run the

entire class or run all classes. This program serves as a makeshift test runner for the framework that is designed to comply to the standards for their tool. The command line tool runs the required setup and teardown methods, and generates nice clean reports for the test runner to read should the test report a failure.

Tactics

There were many challenges to overcome in the testing of the API. One of the most glaring and important was the fact that all of the events in the framework are triggered by Windows events. We looked into a tool used to generate these events, but this proved unfruitful. The only tool that was found was a tool that would make the power management functions return certain values that you want them to return. This would not have allowed us to automate all of our testing because it does not have a way to inject events into Windows so that power status events, like power source change events, will be sent to our API. While this might have allowed us to be somewhat less dependent on running the tests on a computer with a battery, it would not have allowed us to make all the tests run on one computer. This would have been only a partial solution; the benefits were miniscule; and it would have taken a lot of time to integrate this tool into the tests. For these reasons we chose not to use this tool.

We used two different strategies to achieve the appropriate level of coverage. Ideally we would like to have full coverage in automated code; however, this is not possible because of the way the code interfaces with Windows. The two strategies are important to ensure that as much code as possible is covered by automated tests, while also achieving complete coverage. The first strategy was manual testing. A separate application was created that the person running the tests interacts with. The program instructs the test runner to perform activities that will generate events. An example of this strategy is a test that tells the user to unplug the laptop (this test must be run on a laptop) from the AC power to generate a power source change event. When the AC

power is disconnected, it causes Windows to fire a power source change event, which the test waits for. Upon receiving the event, the application reports that the test is a success. If the user indicates that he has removed the battery, yet the test has received no events, the test is marked as a failure.

The second strategy used involves using reflection to simulate the Windows events as close as possible to where Windows interacts with our code. Whenever a test is started, the testing code uses reflection to get the instance of the hidden form within the implementation of our API that is used to receive messages from Windows. Then we call the WndProc method on this form using a Message object that we created to be as close as possible to the Message object that would be sent from Windows.

Implementation

The tests had to be designed to overcome some of the limitations of the Visual Studio unit testing framework. For instance, all the tests are run in the same process, hence any singletons in the API will persist between tests. This is not desired for some tests, because it does not allow us to cover all the functionality. The query tests query for values like: the battery life percentage remaining; if a battery is present; if the monitor is on; and what power source the computer is running on. The API gets these values by registering for change notifications with Windows and then caching the values. If an earlier test had added a change notification handler for the event that corresponds to these values, then the query test would not be valid since it would just be returning the cached value. To circumvent this problem, we created a console application that the unit test launches in a separate process. The console application prints to standard out the value from the query, and the unit test reads the redirected standard out to get the value. It then checks this value against the value obtained from the API which already has the value cached.

We created a number of Visual Studio projects to perform different tasks for some of the tests. Here is the list of projects created for the testing and a brief explanation of their purpose:

1. PowerManagementTest
   This is the main project for the automated unit testing. This is the Visual Studio unit testing project that uses the built in unit testing framework.
2. PowerManagementTestHelper
   This is a console application that is run in a separate processes to query values in the PowerManager class without ever registering for the events.
3. PowerManagementTestRunner
   This is a console application that can be used by Microsoft's test team to run the Visual Studio unit tests from their intranet test manager. This application uses reflection to find all the testing methods in PowerManagementTest and runs them just like Visual Studio runs them.
4. NativeHelpers
   This is a C++ library that uses native Win32 API calls and structures to check the validity of the battery information from PowerManager
5. ShutdownTestSecondProcess
   This is used by the shutdown test to ensure that two separate programs can block shutdown independently of each other.
6. ManualTests
   This is a Windows Forms application that is used for the required manual tests.
7. PinvokeHelpers
   This is a class library that the tests use to provide a common place to access the windows API. Initially each project had the native methods that they required; however, this caused a lot of code duplication and copy and paste issues.

Figure 10 shows the dependencies between all these projects.

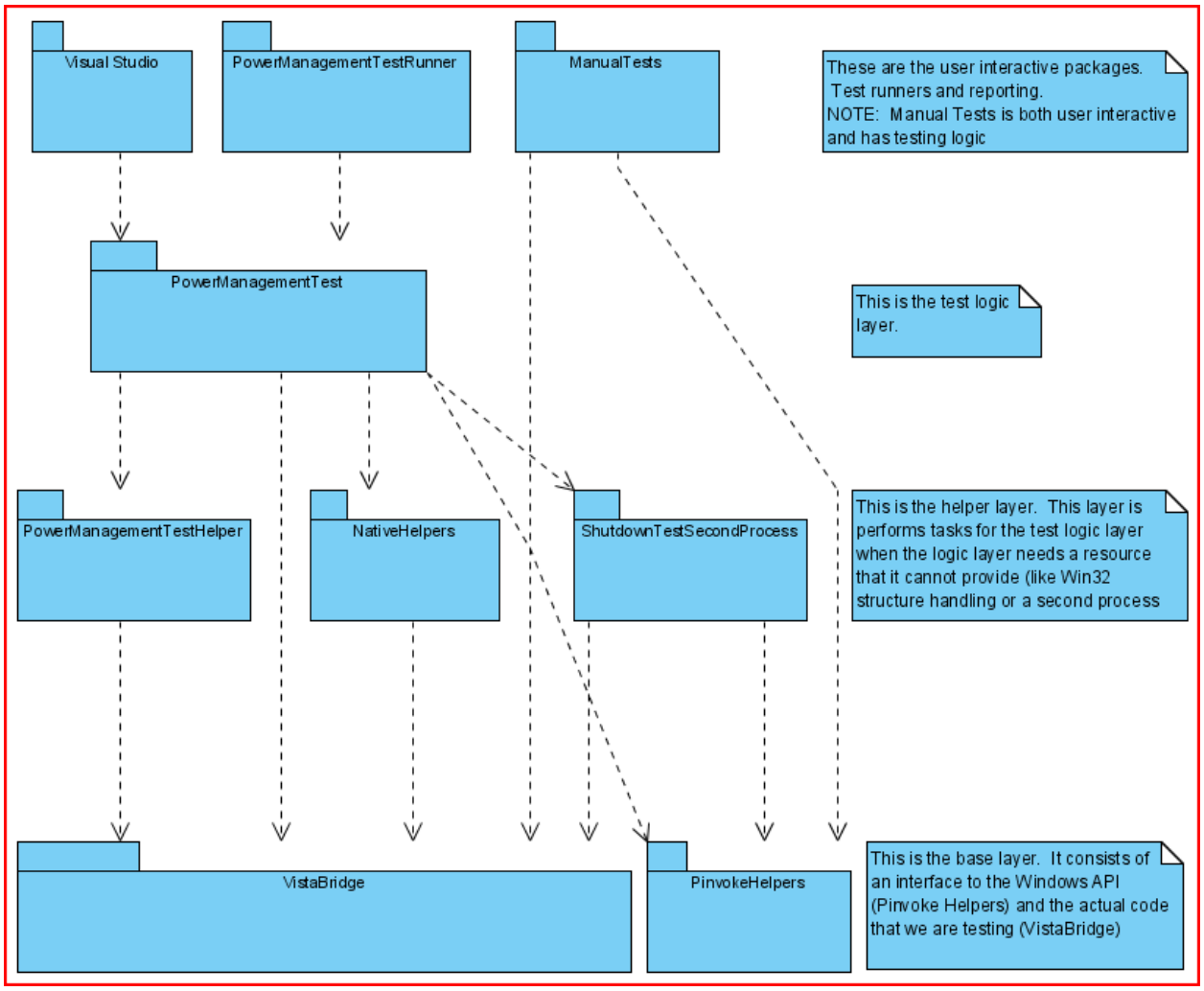

Figure 10

The automated tests in PowerManagementTests cover almost all of the logic in VistaBridge. The only logic not covered by these tests is error handling code for windows API failures which we cannot generate. The automated tests do all the visibility testing, platform testing, negative testing, and most of the correctness testing.

By looking at code coverage only, one could not tell that the automated tests were not adequate. The deficiency in the automated tests is that the testing code is simulating Windows messages to the API, and thus, if the Windows message is different in some way, it could cause the code to fail in an application even after passing the automated test for that exact condition.

For this reason, we created the manual test program that requires the user to take action so that Windows sends the required messages to the program. One of the tests prompts the user to change the power source being used, and then requires that the user verify that the program generated a list of the correct power sources.

Not all of the manual tests are as interactive as the power source change test though. For example, the test for the monitor turning on/off requires no user interaction, but is a manual test, because there must be no mouse or keyboard activity during the test. If the test was run in an automated environment, it could not be guaranteed that nobody was using that computer at the time, and hence there could be false positive or false negative results.

The manual testing is used to cover only the interactions between Windows and our code that could not be tested by the automated tests. This includes tests for keeping the monitor on; keeping the system awake; receiving system busy notifications; receiving power source change events; receiving battery life change notifications; and receiving monitor status change notifications. The power personality events are not tested by the manual tests because the automated testing changes the power scheme which prompts Windows to generate power personality change events.

The manual testing program has a GUI which includes a list of all the manual tests that the user can run. After a test is run, the result is recorded in the application and the test shows up in the list colored coded to reflect the result (green = pass, yellow = inconclusive, red = fail). There is also a text box on the GUI that provides information for tests about why it failed or why it was inconclusive. Tests can only be run one at a time, and while they are running they have access to a panel on the GUI to show controls to the user. Most of the tests provide information as the test is running with a text box, and allow user interaction through buttons. Tests that

require the user to wait for something to happen (like the monitor to turn off after a minute) include a progress bar to show the test's progress. The tests that require the user to wait also change the power settings so that the time to wait is minimized. The program attempts to change these settings back to their original values, however, if the user kills the manual test program with the task manager, there is a chance that the settings will not be reset.

Coverage

Since the manual testing's only purpose is to test the interaction between Windows and our code, coverage information is not important for it. However, the automated tests' purpose is to exercise all the code in the implementation of the API. Calculating code coverage is not straightforward for these tests. This is due to the automated tests requiring a variety of platforms. To run all the unit tests, suites must be dispatched such that all of these conditions are met: the computer does not contain a battery; the computer contains a battery; the computer is running Windows Vista; the computer is running Windows XP; the computer is running a version of Windows earlier than XP.

To calculate the code coverage, we ran the tests on the required platforms and looked at the code coverage. We then merged the coverage result in Visual Studio. This did not yield percentage information, but did give us line-by-line information. We used this information to estimate the code coverage. The only lines, that we found, that were not covered were lines handling error conditions from the Windows API. These lines could not be tested without injecting faults into Windows. There are 11 lines that are not covered, which equates to 2.6% of the implementation. With this information, we feel confident in stating that our code coverage is 97%.

3.4 Example

We created a sample application to show developers how to use our API. The application is a Windows form which displays (and updates when changed) the current power source and battery life. The application also blocks shutdown and counts the number of times shutdown is attempted. Lastly, there are buttons to keep the monitor on and prevent the computer from sleeping. The sample application was also used as a quick demo during our final presentation at Microsoft.

4. Results and Analysis

Our finished product is a complete API usable in any .NET language. The sample application we created demonstrates how easy it is to begin writing "power aware" code with very few lines of code. We gave a final presentation at Microsoft to members of the BCL team, the VistaBridge team (the team maintaining this project in the future), and other members of the CLR group. Overall, everyone was very impressed with our project. The VistaBridge team was very happy with the end result and they believe they will be able to take over the project without any problems.

We have a suite of both automated and manual tests which verify our API is working properly. The automated tests can be run any time a change is made to the code base to verify there have been no breaking changes, making maintenance considerably easier.

The finished project consisted of 3227 lines of code. Of these, 423 were in the API implementation, 2,645 were testing, and 159 were in the sample application. This implies that the implementation was about 13% of the coding effort. Put another way, the ratio of testing

code to implementation code is about 6.2:1. This ratio is high, but is not surprising to us. We further break down the lines of testing code into purpose in Figure 11.

Lines	Use
1334	Automated testing
967	Manual Testing
423	Implementation
344	Code used for automated and manual testing
159	Sample application
3227	Total

Figure 11: Lines of Code

5. Conclusions

At the start of the project, we were not sure which project we would be doing. We were given a list of four projects to choose from, but over time we came to learn that two projects were already completed and one required support from the Windows shell team which we were not likely to get in such a short timeframe. The remaining project was the power management project, which we began working on about three weeks in. Had we begun working on the power management project from the get go, we would've had time to complete or at least start a second project.

Of the design, development, and testing phases, the design phase provided the biggest opportunity for learning. We were able to work closely with members of the BCL team to refine our design. The design meeting for our API gave us some insight into how features are normally designed at Microsoft. It also was an excellent opportunity for us to receive feedback from people of all disciplines. Program managers, developers, and testers all have unique insights into the design of a feature and having them all there really helped shape our design.

Of development and testing, the testing portion of the project was significantly larger than the development. Due to the inability to easily test most power management features, testing became more complicated than usual. While development required hours of thought to ensure maximum performance in addition to thread safety, testing required many hours of understanding and using reflection to simulate Windows messages. This was probably the most surprising aspect of the project and had there been an existing tool to simulate Windows messages, the project would be considerably smaller.

Considering the time spent fabricating the initial design of the API, a main lesson learned on this MQP is to always check with dependent projects before designing an interface to them. Had

we talked with the Windows power management team before designing our API, we could've saved hours of work spent on a design that was not wanted. We were, however, able to work with them for the remainder of the project to ensure our design was only exposing useful behavior.

6. Glossary

BCL – Base Class Libraries

CLR – Common Language Runtime

IL – Intermediate Language

p/invoke – Platform Invoke; Used to call unmanaged functions from managed code.

7. References

1. **TIODE Software.** TIOBE Programming Community Index for November 2007. *TIOBE Software Official Site.* [Online] [Cited: 11 12, 2007.] http://www.tiobe.com/tpci.htm.

2. **Project, The History of Computing.** *Microsoft's timeline from 1975 - 1990.* [Online] April 15, 2007. [Cited: November 12, 2007.] http://www.thocp.net/companies/microsoft/microsoft_company.htm.

3. *Windows history: Windows products history.* [Online] [Cited: November 12, 2007.] http://www.microsoft.com/windows/WinHistoryDesktop.mspx.

4. Microsoft Investor Relations. *Quarterly Earnings Report: Fiscal Year 2007 – Quarter 4.* [Online] [Cited: November 12, 2007.] http://www.microsoft.com/msft/download/fy07/letterhead_Q4.doc.

5. Registering for Power Events (Windows). [Online] http://msdn2.microsoft.com/en-us/library/aa373195(VS.85).aspx.

6. Application Shutdown Changes in Windows Vista. [Online] http://msdn2.microsoft.com/en-us/library/ms700677.aspx.

7. **Cwalina, Krzysztof and Abrams, Brad.** *Framework Design Guidelines.* s.l. : Pearson Education, Inc., 2006.

8. **Michaelis, Mark.** A Unit Testing Walkthrough with Visual Studio Team Test. *MSDN.* [Online] March 2005. [Cited: March 1, 2008.] http://msdn2.microsoft.com/en-us/library/ms379625(VS.80).aspx.

9. *JUnit.org.* [Online] [Cited: 3 1, 2008.] www.junit.org.