"Pushing the kernel on a Swing"

Jane Chu, Sudershan Goyal, James Marks, Saeed Nowshadi, Dirk Ruiz,
Peter van der Linden, Terry Whatley

Affiliation: (all) Sun Microsystems Inc, mailstop MPK28−301,
901 San Antonio Road, Palo Alto , CA 94303−4900, U.S.A.
Email: (all) powermad@positive.eng.sun.com
Tel. 650 786−6409 (office), 650 786−7323 (fax)

Abstract

Many GUIs in use today are so highly coupled to the underlying application that for all intents and purposes, the GUI *is* the application. When a user formats a paragraph in the StarOffice word processor, or browses a new website with Netscape Communicator, the GUI represents on screen every interesting aspect of the data, and provides exclusive control over how it is processed. The term *wysiwyg* is sometimes applied to this type of program. We term this a "high" level of coupling between GUI and task/data, in order to make comparisons with other uses of GUIs.

Other software systems exhibit a *low* level of coupling between GUI and task. An example is the "dial−up" GUI in Windows 95. The Win95 dial−up GUI window pops up unbidden when you invoke any program that needs to make a network connection, such as telnet or ftp. The GUI reports what is happening, but it is an unnecessary adjunct to network operation. Worse than that, the unrequested appearance of a GUI here has a deleterious effect. It forces the program to become interactive, even though the code that invoked it may not have been. If the dialer pops up a dialog box (e.g. to report a busy signal) the entire program can go no further until someone is present to click the "OK" button. But it is not "OK": the user should be able to control whether the unnecessary GUI is wanted or not, and hence whether the program runs needing user input or not. We have here a "low" level of coupling between task and GUI.

There is a third level of coupling between a GUI and an application: medium. A *medium* coupling indicates that the GUI can inspect the application, and control aspects of it, but both can also run independently. Medium coupling is characteristic of well−designed process control GUIs. The example described here is the use of a Java GUI to control the interactions between operating system kernel, and special function hardware. Software developers, particularly application developers, may find this a novel use of Java. A high coupling GUI is easy to write because the GUI is in complete control. A low coupling GUI is easy to write because the application is in complete control. A medium coupling GUI is the most difficult kind of GUI to write because it requires the most careful design of interaction. Concerning this paper, the challenging nature of the medium is the message.

1. Introduction

In recent years, governments around the world have started to make serious efforts to encourage energy conservation in both businesses and homes. Some examples are: the European Commission's SAVE II initiative, Switzerland's E2000 policy, and the "Energy Star" program of the United States government. By conserving electrical energy, consumers can avoid costs, utility companies can postpone construction of expensive new generating plant, and nations can avoid additional pollution and fuel consumption.

In 1997 we discovered that Sun could save about $1M annually in electricity costs in Sun's San Francisco Bay Area facilities simply by aggressive and automatic power management of monitors. You will be hearing a lot more about Energy Star and power management in the years ahead.

1.1 Energy Star guidelines

Energy conservation applies to a very wide range of products, including office equipment generally, and desktop computers in particlular. The latest revision of the US Energy Star guidelines for desktop computers from July 1, 1999 (EPA, 1999) are quite stringent. Part of the requirements can be summarized as saying that, after 30 minutes idle, Energy Star−compliant computers should enter a low−power state where they consume no more than 15% of their power supply capacity. In other words, a computer should notice when it is idle, and turn itself down (or even off) the way modern photocopiers do. The computer must not lose any data when it goes to a lower power mode, and ideally it should restore full power immediately on demand. Most challenging of all, if on a network, the computer should not drop network packets when in a low power mode. We call this requirement being "network aware".

Energy Star support requires cooperation between hardware and software at the lowest levels. It is fundamentally hard to design an Energy Star−version 3 compliant computer, which is why so few manufacturers do it well. However, computer manufacturers are motivated to tackle this difficult task because US government desktop computer procurement is required to prefer Energy Star compliant equipment.

Many Microsoft Windows laptops support Energy Star, because you get the difficult parts of the hardware design for free as part of trying to eke the longest life out of batteries. Many desktop systems that run Microsoft Windows do not support Energy Star. No Linux−based system has support for Energy Star. No Unix−based system, except Solaris, has support for Energy Star. All desktop computers from Sun Microsystems are Energy Star compliant and have been for the last 5 years. "Power Management" is the Solaris kernel subsystem that implements support for the Energy Star rules. The authors of this paper are the kernel software developers responsible for power management within Solaris.

In 1998/9 Sun worked on a new power management framework to comply with the new, much stricter, version 3 Energy Star rules. We needed to update our existing GUI to work with the new power management framework. The remainder of this paper describes the GUI application that we designed and wrote for that purpose, and what we learned in the process. The GUI was written in Java, and made extensive use of the Java Foundation Classes, also known as the Swing library. As far as we know, this is a unique use of Java, to provide user control of a hardware−level Solaris kernel subsystem. We are pushing the kernel on a Swing GUI.

1.2 What the Power Management system does

The goal of Solaris power management is to intelligently match the power consumption of a system to its level of use, and to meet the Energy Star guidelines. When the system is idle for a (user−configurable) period, the Power Management (PM) framework should reduce system power consumption.

In general there is a tradeoff between the state−of−readiness of a peripheral, and its power consumption. You can cut the power to disks or DVD drives when no one is using them, but it will take a few seconds to spin them up to operating speed when they are again needed. This must be done automatically, with no special command from the user. There may be several levels of decreased power use, culminating in "off". Later the PM framework must bring the computer back to its previous user interface state in the shortest possible time. All this must be done while introducing the least overhead to normal system operation.

1.3 PM is a complex system with detailed hardware and kernel knowledge

The Power Management kernel subsystem on Solaris works by keeping track of how hard the system is working. It then selectively requests device drivers to move hardware to higher or lower power states. It is an "industrial strength" framework that provides full generality to system designers.

The ACPI approach to Power Management used in the PC world makes tradeoffs which provide an easier implementation, but one which is much less robust for users. For example ACPI relies on priming the network interface with a "magic packet". The "magic packet" is a list of IP packets that will cause the Network Interface to wake up from low power mode. (You don’t want to wake up for every IP packet, or you risk spending all night continually cycling between awake and asleep, depending on how much broadcast traffic is on your net). NIC designs for PCs only hold a limited number of magic packets that will cause wake−up. So the ACPI approach does not scale up to environments that make extensive use of networking. That was unacceptable to us, and we designed a PM framework that avoids this kind of limitation. Unlike PC power management, we further support permissions−based capabilities, and remote administration. Our implementation allows a system administrator to define and apply a department−wide PM policy, but also allows users to fine tune power management.

Current hardware devices, from ASICs to buses to disks or monitors vary in their support for power management. For those that have power management features, the hardware implementations often vary considerably. Newer frame buffer cards (video cards in PC terms) typically support several power states namely *On*, *Standby*, *Suspend*, and *Off*. Older frame buffers only implement a subset of the four states. Monitor devices have the same range of capabilities. PCI host buses of 33MHz speed that comply with the PCI specification v2.1 implement four power states: B0, B1, B2, B3: which correspond to *full power*, *idle with state preserved*, *clock off with state lost*, and *power off* respectively. For devices with more than two power states, power transitions between different states have to follow a state diagram which is usually only meaningful to that particular device type.

Besides power state, there are device inter−dependencies to be considered. For example, since the frame buffer drives a monitor, it cannot be driven to a lower power state than monitor. In order to lower the power of a PCI bus, all devices attached to that bus have to be switched to a lower power

mode. System idleness before switching to low power mode, the check of the number of power cycles executed against devices's maximum life power cycle allowed, the optimum approach to perform power transitions for multiple component devices, etc. − all have to considered by the Solaris power management frame work and the device drivers that support it.

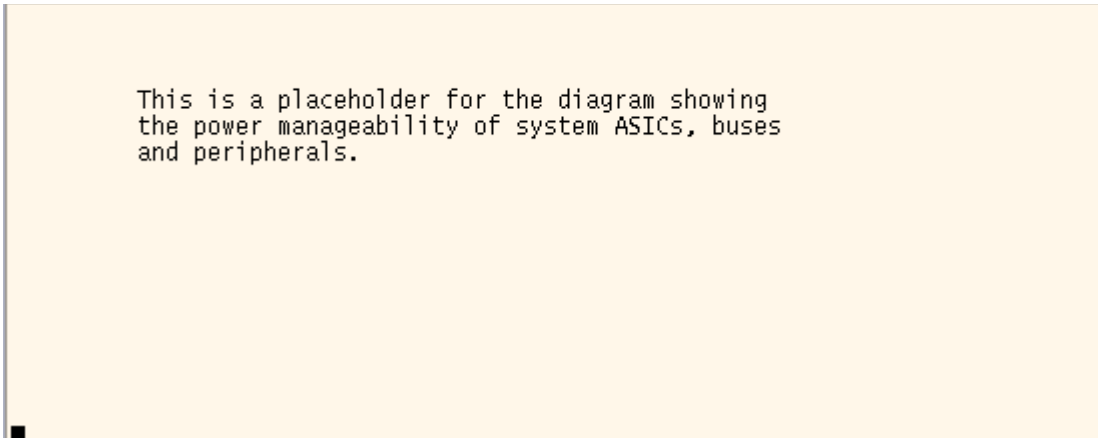

Figure 1: block diagram of power management possibilities in a system.

The new Solaris 8 power management framework provides an efficient and dynamic software means to regulate the power consumption of a system. Sophisticated users can choose what devices they want to be power managed, and the various idle periods for them. For example one user might choose to power manage the monitor after 20 minutes idle, and the disks after a further 10 minutes. Another user might be so enthusiastic about energy conservation that they want to power manage everything to save the most power after the system has been idle for 12 minutes. But it is an unreasonable burden on the average user to ask for an understanding of buses, device drivers, and peripherals in order to configure his or her desktop for the desired power management. So an additional challenge for a GUI is to make it simple enough for anyone to use. We did this with carefully considered defaults for common operations.

1.4 Objectives for the new Power Management GUI

We set ourselves four clear goals for the software development. In order of highest priority first, our goals were:

1. To evolve the power management GUI to match new Energy Star features.
2. To make the GUI easier to understand and use.
3. To make the GUI code more maintainable.
4. To let users do everything they could do with the old version of the GUI

To expand on the third goal (make the code more maintainable), the older, Motif−based version of dtpower was ridiculously difficult to update and sustain. It was written in C, and made extensive use of pointers and manual memory management. The last change made to this Motif code was programmed by the team manager, because all the programmers had more sense than to wrestle with it. With Swing, Sun has at last created a user interface library that programmers really seem to like, and are learning in large numbers.

The most minor (last) goal was probably the hardest, both from the design perspective (where does

that new code belong?) and from the user's perspective (how can users be helped over the surprise of so much new functionality?). Backwards compatability is an extraordinarily confining strait−jacket which has to be tried on to be fully appreciated.

Finally, yes, we admit it, as highly experienced kernel C programmers we wanted to take the opportunity to engage with GUI programming, and successfully write a modern GUI using Java. The software developers didn't take any formal training in Java or Swing; we felt we had enough experience to learn on the job, and we were right.

1.5 Size of the application

Table 1 below gives some key statistics of the new "dtpower" application we created. The "dt" in "dtpower" means "desktop". Many applications in the CDE window system have names that start with dt, like dtmail, dtcm (calendar manager) and dtpower. The application can be invoked from the command line via:

```
/usr/dt/bin/dtpower
```

It can be invoked from the CDE window manager by clicking on the background to bring up the workspace menu then selecting "Programs" −> "Power Manager".

Total source lines:	8300 lines of source
Total Java code:	5600 lines
Total C language code:	1000 lines
Total makefiles, help documents, etc:	1700 lines
Development & test effort:	25 person−months

Table 1: Some development statistics

Note: team members also had additional responsibilities (code maintenance, other development etc) while working on this project, so productivity metrics cannot be directly applied to these figures.

2. Methods and Approach Taken

The software development team partnered with Sun's Human Interface Team to design the new user interface. There were three reasons for involving GUI design professionals early in the process. First, at the very least, an HIE review is required to satisfy Sun's internal development process. Second, by sharing the design work, the HIE team lifted some of the project burden from the shoulders of the software developers. Finally, the HIE contribution most assuredly made for a better final product.

2.1Start with help from Human Interface Experts

Almost every design choice in a GUI boils down to a matter of opinion that can be (and is) passionately discussed at great length. In the end, software developers need to accept that good interface design requires effort, experience, and knowledge of users that most of us just do not have. While some software developers may be expert UI designers, we have not met them yet and this experience is mirrored elsewhere (Cooper 1999). HIEs have expertise in a variety of domains e.g. GUI standards, typical user models, and usability testing methodologies. A product developed without the application of these skills may not be a guaranteed failure, but it risks being of much

lower quality than desired (Cooper 1995). For examples of the kind of GUI antics you get without HIE help, see (Interface 1999).

While we acknowledge that programmers don't generally know all that much about GUI design, it's also true that Human Interface experts generally know little about power management. Each team member brought an expertise, and also the willingness to learn from and defer to the expertise of others.

2.2 Refining Goals into Design Requirements

A core team was assembled. The core team started with only three people, but later grew in size. One person brought a broad understanding of all EnergyStar issues, one person was assigned to do the actual coding, and the third person was the Human Interface engineer. The first task for the team was to expand the requirement statement of "write a GUI for power management" The team held regular meetings and, after extensive discussion, reached agreement on the following:

- The primary focus of the GUI was to provide user control over power management parameters, rather than the easier task of simply reporting the power management state of a system.
- The GUI should serve all types of users. It should work for naive users who want simple tasks like "disable power management" or "switch to default power management mode". It must also service advanced users who want to power−tune each component of every device on the system.
- The user should be able to control all the parameters using the GUI, and should never need to resort to editing configuration files.
- The GUI must hide the different sources of power management activities on the system. For example, the kernel controls all devices except framebuffers and monitors, which are controlled by the Xserver (network window manager). The power model used by the kernel is remarkably different than that used by the Xserver. The user doesn't care about or need to know about this kind of implementation detail. The GUI will not expose the user to this type of confusion.
- The GUI will work across all hardware supported by Solaris. SPARC servers support monitor power management only. SPARC workstations support different power management schemes depending on the version of the Energy Star guidelines that they comply with. Solaris on x86 runs the framework, but the x86 group in Sun hasn't put power management support in their device drivers yet. The GUI will understand all these differences, and present a consistent picture to the user.

Based on these requirements, the Human Interface engineer created many drafts of possible GUI designs. Many were quickly discarded. The remaining designs were refined in regular meetings over the course of many weeks. During those meetings each and every item on the layout was judged on the functionality it provided and the complexity it added to the design. Diagram 1 shows the final result of the process.

This paper isn't a manual for PM, so we won't walk through it in detail. The main point to take away is that it took many weeks to arrive at a clean, simple consistent design, and we visited every other combination of possibilities on the way there! Simplicity isn't something you put *into* a design; you get there by methodically taking the complexity *out*.

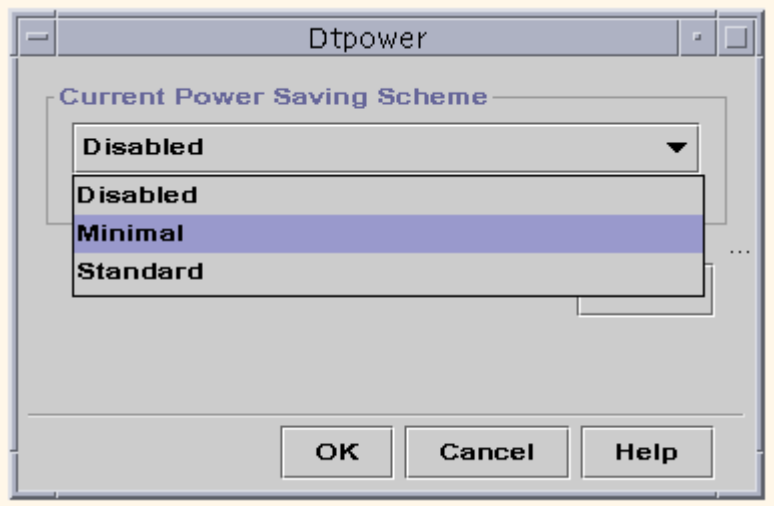

Diagram 1: The basic Power Management GUI.

When the draft design was completed, it was reviewed by everyone in the group and shown to upper management. This resulted in more changes in the design. The main change was further simplification. The core team had discussed power management every week for many weeks, and we lost sight of the fact that some of the things we thought were trivial were in fact difficult for the people who don't know much about power management. Showing the design to people with less power management context revealed these flaws.

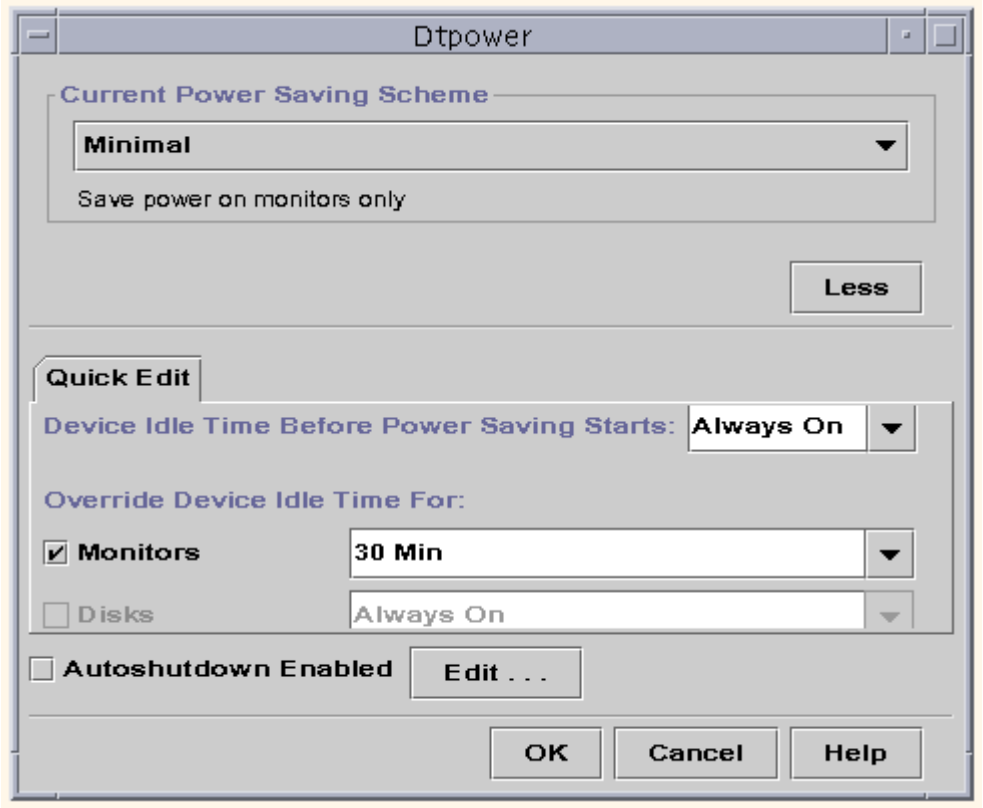

Diagram 2: The expanded dtpower GUI

So far the English text on the screen was provisional. We realized that we needed the help of a professional technical writer when people started to deconstruct different meanings of even simple labels like ’Default System Idle Time’. Some people thought that the Default was with System and others thought that Default was with Idle Time. After *many* iterations on the wording ("idle time before a device shuts down", "idle time before power managing a device", "default idle time before power management", "idle time for all devices before saving power", etc) we finally settled on:

Device idle time before power saving starts

which is not strictly true, but close enough. The strictly true label would be something like "Device idle time before you reach the final power state of which this device is capable". And this is just the English version. Imagine what you get when you translate it into Swedish. This is the process for just one text description on the gui; they are all like that!

2.3 IDEs: programmer’s boon or bane?

After the GUI design and screen layout was almost final, the actual coding began. Given the amount of work, we quickly realized that we couldn’t finish the entire task in the allocated time. It is not the coding that is the problem. Coding in Java is fast, and you don’t spend any time fussing with memory management or the minutiae of networks of data structures bound together with pointers either. Where you spend the time in software development is all in the back end process of testing, code integration, process review, localization, document writing and review, etc. This is why two−coders−in−a−garage can produce phenomenal amounts of code, *if* they don’t have to do any of the things to productize it. The back−end load resulted in our decision to implement the GUI in two phases. Phase I consists of all the functionality needed for the beginner and the intermediate user. Phase II is the functionality needed for the advanced user. Hence the tabbed pane with one tab seen in diagram 2. We plan to add more tabs later, and wanted to build that expectation in now.

To get the project going we used an IDE containing a GUI−builder. It was invaluable when we were prototyping and attempting to converge on what we actually wanted the final product to look like. This was the first Java project and first exposure to the JFC (Swing Toolkit) for the developers. So there was a side−benefit to the IDE: it served as a code tutor. By dragging and dropping components from a palette onto a form, we could look at the generated code for a model on how to program this ourselves should the need arise (which it later did).

At least as important as the RAD features of an IDE was the debugging support. The ability to set breakpoints at the source code level, single−step while watching source, examine variables at any point in the class heirarchy all helped us to walk through all the important paths of our logic.

3. Approach used

Our application needs a list of all devices connected to the system. We use the JNI to develop such a list. The native code opens a special device (called "/dev/pm" − all devices can be accessed as files in Unix) which acts as the interface to the kernel for power management operations. The code must walk the entire device tree and for each device, it must query the kernel using an ioctl system call to find out whether the device is power manageable.

We build this the obvious way. During initialization the Java code instantiates a `SystemInterrogator` object. The corresponding class is implemented in C and accessed via the JNI. The C code opens the special pm device and makes the ioctl (Unix I/O control system) calls to

build up a memory−resident data structure of power management information. This all happens as a result of calling the constructor of the SystemInterrogator class.

Once this is done, the Java code can call a variety of functions exported by SystemInterrogator to retrieve selected data items which describe the pm capabilities of the machine. One of the calls made by the Java code is to the function getDevices() which returns an array of Device objects. A Device object exists for each power manageable device on the system and contains all the relevant information about the underlying device.

In the early phases of the project, some of the newer power manageable devices were not available. To keep the forward momentum of testing as much logic as possible, even before the arrival of hardware, we made the Java class that asked about hardware also generate dummy Device objects populated with likely data. The use of stub objects while waiting for more code is a time−honored software technique.

3.1 Java Native Interface

In order for any application to access the power management features of Solaris, it must talk directly with the operating system using system calls. The first requirement is to determine what level of power management support is available on the current machine. This is necessary because older workstation models offer only the coarsest means of modifying power consumption: utilizing the built−in pm features of the monitor, spinning down disk drives, or preserving the state of the OS and powering down the entire machine.

We had no inhibitions about using JNI to make system calls. Power Management is tied to systems that run Solaris (currently SPARC and Intel x86), so WORA was not a goal for this project. There are many benefits to Java other than platform independence: simplicity, rich libraries, good object model, automatic memory management, etc. As it happens, the same GUI bytecode does run on Solaris and x86, and it calls out to ported versions of the same C libraries.

We knew that crossing the JNI barrier is expensive in time. That didn’t matter in the context of a GUI. You would want to minimize the number of traversals if you were doing something like 3D processing. We found the Java Tutorial and the book by Sheng Liang (Liang 1999), the designer of JNI, to be valuable resources in teaching ourselves JNI. Being inside Sun did not bring us any special learning edge or access to non−public resources. Sun is a very large company, and the time when all the software developers in the company could meet in a single conference room for milk and cookies ended in about 1987.

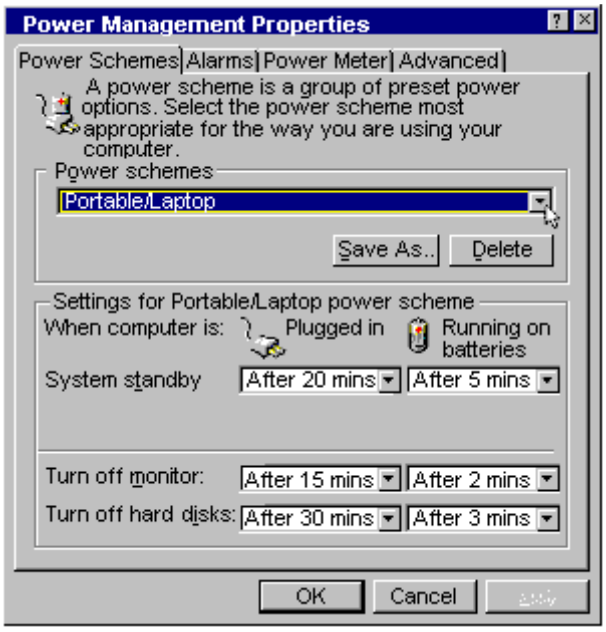

Diagram 3: Windows 98 power management GUI. Our design looks similar, but allows more sophisticated control of the hardware. We feel our design also does a better job of presenting the complexity gradually. The Windows GUI betrays its origins in laptops, with a reference to running on batteries.The Windows GUI uses small icons in a nice way. We will consider that for our next update. Small visual touches can make a GUI more approachable, if used carefully.

Sun's CDE (Window manager) group helped us with advice on global issues like layout of source so that our GUI is consistent with other CDE Java GUIs. We also gained their assistance in integrating the GUI with CDE so that it appears in the appropriate menus and is invoked when the user clicks on the icon. The point here is that software development within a large existing product (Solaris) requires effort and cooperation from several key groups that are outside your own department. We find that management can help things go smoothly here.

3.1 Debugging

Some moments of weirdness stand out with respect to debugging. One was the discovery that programmatically selecting a JComboBox selection (with setSelectedIndex()) actually fires an event just as if the user had selected that entry with the mouse. However, programmatically selecting a JCheckBox (with setSelected()) does *not* fire an event. Why do the two controls have this apparent inconsistency? It was because they were written by two different programmers, who made different (reasonable) assumptions. Anyone can report bugs against Java using (1999 javadev), and is encouraged to do so because developers only fix problems they know about. We filed bug 4296747 against this.

In a similar vein, we noticed an inconsistency between the behavior of a JTextField when used as a component directly on the window as opposed to its behavior when used as the editable entity of a JComboBox. In the former case, a user can type data into the text field, but if the focus is

transferred elsewhere with the mouse (as by clicking OK to dismiss the GUI), no event is fired for the text field. And the behavior turns out to be the opposite for a JTextField used in a JComboBox: a click on OK will cause a JTextField to fire an event. All of these behaviors violate the "Law of Least Astonishment" for software, and they represent an unnecessary loss in productivity as individual programmers discover each of them the hard way.

As convenient as IDE debuggers can be, neither of these problems were discovered and solved until after we had migrated away from the IDE. Just as printf() is the C programmer's best friend, System.out.println() can be the Java programmer's best friend.

Most of the implementation problems came not from dealing with a gui, but from the system interface and the semantics of the power management framework itself. Our design requirements imposed these needs on our GUI:

- having to launch another application to do chores which require root permission,
- carrying along an existing framework because of backward compatibility (and the attendant compromises therin),
- having to test at startup that the user hasn't sneakily modified any configuration files so that the system is no longer running the scheme the gui thinks it is,
- configuring the initial state of the gui's controls based on whether it is an Ultrasparc I or II, a legacy 4m, or an Estar version 3 compliant machine,
- hiding from the user the fact that monitors are managed by a different subsystem than all the other devices,
- coping with OS features that would be useful to many, but do not currently exist. E.g. there is no safe way currently to map from a pathname in the device tree, to the kind of device it is (disk, screen, etc).

As part of encapsulation, we want to minimize the amount of kernel−specific and hardware−specific knowledge that spills into the GUI. But we have to give the GUI enough information to do its job. The issues with trying to enforce a permission system means that the gui must decide on which controls (if any) a particular user is allowed to operate, and indeed, which schemes he is allowed to select. Inappropriate controls must be grayed out and disabled.

3.3 How to code/test i18n when you don't speak Korean.

The term "i18n" is an abbreviation for "internationalization" − the process of writing your code so that the natural (English) language messages will all be loaded at run time. Then, later you can easily localize the program to a different language (French, German, Japanese etc) by plugging in the appropriate localized (translated) file. For example, to i18nize the "Hello World" program in Java, you would read the string to be printed from a file whose name includes the name of the locale the that system is using. Thus, in a different locale, the program looks for a different file. If the file is present it contains the localization of each string in the program. It is not just literal strings that must be localized. You must also take care of button labels, help files, date formats, number formats, tool tips, and of course, any input of dates, currencies, and numbers in locale−specific formats.

The process of internationalizing our GUI was more work than we originally anticipated or budgeted. The ResourceBundle approach used by Java is not to blame; we found that to be quite

natural and easy to work with. It is superior to the "dialog editor" approach to i18n used by Microsfot's WFC. The dialog editor is most suitable for code with a very small amount of i18n, and doesn't support any way to carry the localizations forward to the next version of the software, Our problem was more one of logistics and vigilance. In our case lack of vigilance caused us a little embarrassment.

The logistics issue comes from the fact that every group in a large software factory needs a certain time to perform all their work once they receive all their inputs. By the nature of the software development process, the folks who do the actual translations for the l10n must be at the end of the chain. And, in the case of error messages which need to be translated, developers may not know until beyond the end of the normal development phase what all the messages are going to be. E.g. late testing uncovers the need to display messages like "*Do not pull the chain while train is standing in station*". Adding a new message can lead to a lot of pressure on the people who do the localization and the testing. There is usually very little time for adding new translations.

The lack of vigilance referred to above occurred because we developers did not concern ourselves with i18n during the first 90% of the development process. (And we would bet dollars to doughnuts that almost every other programming team starts out the same way). All text was simply hard−coded as traditional string literals in double quotes, or in some cases as a "public static final String". When it came time to convert these to `<resourceObject>.getString()`calls, the job was done quickly. Unfortunately, a couple of critical messages were missed. Given the work schedule of the localization team, it was difficult to get these messages fixed in time for the release. We managed it, but we don't want to do that to our colleagues again. And it is surprisingly costly to localize a program. It cost about $30,000 (October 1999) to translate all our text from this one small GUI into the 9 main locales that Solaris uses (English plus, French, Italian, German, Spanish, Swedish, two kinds of Chinese, Korean, and Japanese).

A final bit of trickiness regarding i18n came from the fact that in the "scheme" combo box on the Basic screen, the choices are not hardcoded in the source of the program, but actually come from the filenames in a special directory. In other words, each item in the combo box's list is really the name of a file in that directory, one per scheme. Each file contains a text string with a more complete description of the purpose of the scheme. Both the filenames and descriptions were, of course, in English. We could think of no better approach than to have a look up table in the Java program which contains all the known name and description strings and look them up in real time when the corresponding scheme is selected by the user. Having identified the correct name we could then invoke `<resourceObject>.getString()` with the appropriate key.

4. Results

4.1 Human Interface Perspective

The single regret from an HIE perspective is the lack of usability testing. No matter how good the UI designer, there is simply no replacement for putting the product in front of real users, and watching as they try to do a series of tasks. It is almost always the case that improvements will be needed. Usability testing was not done in this case because of HIE resource prioritization. Obviously you should spend HIE resources according to how frequently a GUI will be used, and we need to stop gilding the lily at some point.

Turning now to the cost/benefit analysis, there were several costs to the process that was followed. Time was needed for the HIE representative and the project developers to educate one another. Second, the design work itself took time. Long design sessions can sometimes add to the pressure of a hurried project. Third, the process used requires developers and HIEs to clear product changes with one another. This kind of consensus−based process can sometimes be slow. However, following this process had certain benefits. A well−designed user interface has many of the same characteristics of well−architected code. First, modifications are easy to make, thus significantly reducing the pressure of the last−minute crises that plague every project. Second, it is much easier to communicate a well−designed user interface to others. This includes technical writers, managers, and other developers or HIEs who have to work on the project. Finally, the team members themselves, having bought into an overall vision, are much more likely to be able to do "the right thing" without having to be explicitly told what to do. For example, a developer may be able to implement a lot of functionality without having to ask the HIE frequent questions.

Development and HIE need to be a real, integrated team. This means that they must take the time to educate one another. HIE must educate developers about what they do, and what they know of target users and their needs. Developers must educate HIEs about technical aspects of the domain, and how their development process works. Afterwards, both developers and HIEs must strive to be as close a team as possible. A compartmentalized process, in which HIE throws designs through the transom and developers throw code back, simply will not work. Design is hard work; it requires creativity and trust on both sides to happen.

4.2 Software developer perspective

Our first and most obvious conclusion is that Swing is a rich library, full of useful controls (widgets) that were late arrivals to the Motif library. Controls for displaying a Tree, a TabbedPane, and an ImageIcon only appeared in Motif 2.x, years after Motif was first released. A tree is the natural data structure for so many things in computer science that you wonder how a GUI toolkit could limp along without it. And Swing isn’t all sizzle and no steak. Unlike the Microsoft Foundation Classes used to program GUIs on Windows, Swing has a rich object−oriented model, making it easy to extend controls with new behavior. The Model−View−Controller design pattern is a fundamental part of key Swing components. It allows the programmer to separate concerns which tended to get mashed together in prior models. The downside of MVC is extra effort required on the part of the developer to learn a new design pattern (GOF 1995).

We experimented with certain components which seemed quite natural for the task in hand, but dropped them for reasons of simplicity and deadlines. For example, although we wrote all the code, we decided not to include the JTree to display a device tree in our current version of dtpower. The Jtree displays the all the peripherals and buses attached to a system and allows the user to click on the node representing a device to access a menu and customize the power management for that device. Although the code was written, we concluded that the testing, debugging, localization and documentation needed would cause a schedule miss. So we simplified the implementation to meet the fixed schedule. You must give yourself this kind of flexibility to be a successful software developer, and you *must* keep your management well educated about it.

One problem commonly experienced with the JFC is that it can take a few seconds to get the first image of your JFC−based GUI to appear on the screen. A lot of smart people are hard at work inside Sun to increase the "snappiness" of a Swing GUI start−up, and we hope there can be a

noticeable improvement. But if program start−up is slow off the mark, programmer start−up is rapid. It would have been very difficult for the EnergyStar team to learn Motif GUI programming in the short time allocated for the GUI development.

The issue of whether to use an IDE and a GUI−builder is a thorny subject. At the start of the project, as novice gui−programmers we were mesmerized by how quickly we could put together a very reasonable looking protoype. It was very easy to make modifications to the prototype. But at a certain point in the project, the GUI builder became more hindrance than help. Some of the code it generated looked ugly, and occasionally, unnecessary. One of the highly−touted features of these tools is that they are capable of taking modified source code and back−parsing it into the internal notation which is used to display the objects originally dragged and dropped onto the form.

However, we soon found that living within the type of modifications the builder would allow became too burdensome. Sometimes, it wouldn't allow us to do things which produced better looking, more professional code. At other times, we could not find a way to get the functionality we wanted by using the builder. An example of this was our wish to respond to a button click by splitting one table into two tables occupying the same space. We programmed that easily manually, but could not find any acceptable way to persuade the GUI builder to do it for us in the first place, nor to buy back our code.

We brought much of this grief on ourselves. When, after a long cycle of working with the Human Interface people, we finally settled on what the screens should look like and how all the controls should function, we should have set aside the code which came from the gui builder. Nothing so drastic as throwing it away; it would have been useful as a reference resource. Nevertheless, we should have started coding from scratch so the code could look exactly the way we wanted it to and we wouldn't have wasted time trying to write back−parseable code.

Future Plans

Some cars have a needle gauge that shows the tradeoff between performance and fuel economy at any instant. Some have a microprocessor that can give you the same information in digital form. One enhancement we envision is to develop a "reporting" feature for PM that represents the system state in terms of performance vs. energy economy at a given moment. This could be reported simply by animating a gauge on the dtpower icon. Or it could be a more comprehensive display on a separate panel within the GUI, including historical data and estimates of actual financial savings.

All phase II functionality will be developed eventually. However, before that is done, we will get user feedback on our initial effort. User feedback can come from a variety of sources; most likely, this project will use emailed customer feedback and usability testing. Emailed customer feedback will be collected and categorized to extract the main themes. These will provide ideas for redesigning the user interface. Next, usability tests will be conducted. Depending on time and resources, users will be presented with either the current system, or with the phase II system. They will be asked to do a series of tasks, and to think aloud while doing so.
Task performance and think aloud data will be collected and analyzed to find out what is proving to be the most (or least) confusing aspects of the interface. Armed with this data, a series of design modifications may be proposed.

References/Bibliography

(Cooper 1995) "*About Face*", Alan Cooper, publ. IDG BooksWorldwide 1995, ISBN 1568843224.

(Cooper 1999) "*The Inmates are Running the Asylum*", Alan Cooper and Paul Saffo, publ. Sams 1999; ISBN 0672316498.

(EPA 1999) "*Energy Star−labeled Office Equipment*", http://www.epa.gov/appdstar/esoe/computers.html, webpage published by Environmental Protection Agency, 1999.

(GOF 1995) "*Design Patterns: Elements of Reusable Object−Oriented Software*", Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch, book published 1995 by Addison Wesley Publishing Co., ISBN: 0201633612

(Interface 1999) "*Interface Hall of Shame*", http://www.iarchitect.com/mshame.htm webpage published 1999 by Isys Information Architects Inc., 5401 McNairy Drive, Suite 100, Greensboro, NC 27455, email: bchayes@iarchitect.com

(Javadev 1999) "*Report a bug or Request a Feature*", http://java.sun.com/cgi−bin/bugreport.cgi webpage published 1999 by Sun Microsystems Java Developer Connection, 901 San Antonio Road, Palo Alto CA 94303.

(Liang 1999) "*The Java Native Interface: Programmer's Guide and Specification*" Sheng Liang, book published 1999 by Addison−Wesley Publishing Co; ISBN: 0201325772

(EETimes 1998) "*Power Management Hears a Wake−up Call*", Electronic Engineering Times, magazine feature, issue 1024, Sept 7 1998, page 1

6. Still to do on this paper.

- Final formatting, and page breaks
- Remove duplicate spaces
- final spell and grammar checking, etc.
- All diagrams and most screen dumps
- Final list of references and bibliography
- Incorporate referee comments
- Review for trademarks etc by Sun Legal.
- Review by marketing