

Software Design Patterns Applied to Physical Object Devices

by

Darren V. Hart

A thesis proposal submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2003

1 Introduction

Designing and implementing networks of embedded devices is currently a complex cross-disciplinary undertaking, involving software architecture, embedded programming, and custom electronics. If these hardware devices could be treated as software objects, the problem would reduce to the application of good software design principles and techniques. The degree to which hardware devices can parallel software objects is not clear nor is the set of requirements object-oriented design places on objects. By determining this parallelism and implementing the required features in a set of hardware devices, we can greatly simplify the design of such networks.

1.1 Motivation

Different disciplines use different processes, methods, and strategies to accomplish their goals. The level of cross-discipline interaction, especially involving computers and embedded systems, is increasing. Non-programmers often find themselves learning more programming than they would like to accomplish their goals, and experienced application programmers often find themselves dealing with the low-level interfaces of integrated circuits and designing some amount of custom circuitry.

Part of what makes professionals efficient is their ability to master the processes and methods used in their field. Learning these patterns can be time consuming and becoming proficient with them can often take years of experience. Clearly, cross-discipline projects have the potential to be very time consuming as the developers have to at the very least familiarize themselves with the development patterns of several disciplines. In some situations, the developers are unwilling or unable to learn the skills of other disciplines, Dr. Dan Olsen (Computer Science professor at Brigham Young University) said once in an interview with the author, “Soldering is infinitely hard”, a sentiment hardly conducive to any low level embedded device development. There are many projects that involve incorporating a network of embedded devices to produce an aggregate device, such as a control network, any of several robotic applications, or something less obvious like a DVD player or a home media network. All of these examples involve programming, or may be the means to explore new areas of software (the mechanical robot provides the physical component of AI research, interior mapping, etc.).

Computer scientists make use of software design patterns, a set of generic solutions to several classes of problems, in their design of new software packages. The singleton is one of

the most commonly used software design patterns; a singleton creates an instance of itself at the first request for instantiation and returns a pointer to it for each subsequent request, sharing the same instance with all objects. A singleton provides global access to the object and guarantees that only a single instance can be created. If programmers could apply the development methods they are already proficient with to the design of embedded device networks by treating each piece of hardware like an object, the proven benefits of object-oriented design and patterns would apply to these projects as well. These benefits include faster development times, more easily maintainable code with fewer bugs, and smaller codebases.

Object-Oriented Parallels in Software and Hardware

Both software objects and hardware devices receive commands and return results. Software objects do this via member methods while many hardware devices do this through a device specific wire protocol. While software uses pointers and references to access objects, the hardware objects typically only provide one physical connection which inhibits inter-device communication. In order for programmers to be able to design to an object-oriented API with the familiar patterns just mentioned, these hardware devices must behave like objects. The extent to which hardware objects parallel software objects must be determined, as well as the functional requirements these patterns commonly place on software objects. The appropriate intelligence must then be implemented in the hardware devices to allow them to behave as software objects to the extent required by these design patterns. We will refer to the resulting devices as Physical Object Devices, or PODs.

PODs will provide the perfect platform to explore the use of software design patterns applied to embedded device networks. PODs will be embedded devices (often involving sensors or actuators) that perform some function and can communicate with other PODs on their network. PODs may monitor temperature, provide a control panel, display text or images, sound a buzzer, or perform any other task an embedded device is capable of. The capabilities of each of these PODs will define a new class of POD and each physical POD is analogous to an instance of a POD class. Applying software design patterns to this set of classes and objects facilitates their use by computer scientists already familiar with the patterns.

1.2 Software Design Patterns

Skilled software developers have mastered, or are focused on mastering, software design patterns that aid them in developing modular, reusable, easily maintainable designs. Design

patterns are a well known set of generic solutions that can be applied to specific classes of problems. With design patterns, developers can often find a solution by accurately defining and classifying their problem, rather than having to develop a custom solution. Design patterns are structural, behavioral, and creational in nature.

“Structural patterns are concerned with how classes and objects are composed to form larger structures” [GHJV95]. Structural patterns such as the Adapter, Facade, and Decorator enhance object communication and functionality. The Adapter allows a foreign interface to be mapped to the interface used by the system. A Facade provides a common object through which other objects can communicate with a group of related objects managed by the Facade. A Decorator wraps an existing object to add functionality and responsibilities to it without modifying its external interface.

“Behavioral patterns are concerned with algorithms and the [communication and] assignment of responsibilities between objects” [GHJV95]. The Strategy pattern uses objects of a common base class to carry out a specific task in a variety of ways, such as text layout, memory allocation, and integrated circuit routing algorithms. Signals and Slots are used to provide a powerful and flexible event handling system, signals are emitted by an object when an event occurs and each slot pertaining to that signal is executed. Slots are often bound to methods of other objects.[Gre03]

“Creational patterns abstract the instantiation process” [GHJV95]. A common example of a creational pattern is the Singleton class. A Singleton class creates an instance of itself at the first request for instantiation and returns a pointer to it for each subsequent request, sharing the same instance with all objects.

Design patterns make extensive use of object references, inheritance, and composition. As such, their implementation will depend on these and other facilities being available in the objects it incorporates. We will explore the application of several design patterns to networks of embedded hardware devices to determine what these facilities are and comment on the general applicability of patterns to these types of networks. Some patterns will obviously be more applicable than others.

Creational patterns are likely to be less useful in hardware than in software, since the hardware devices already exist. Structural patterns and particularly behavioral patterns should prove to adapt well to hardware. For example, Frank Sorenson is also doing research with PODs

and has created a few simple devices using USB and an API of his own design. His PODs will certainly not work out of the proverbial box with our infrastructure. Let's say we decide we need a temperature sensor, Frank has already designed, built, and tested one, so we would like to reuse his. The Adapter pattern can be applied here to use Frank's PODs with our POD network. The Adapter translates commands from the POD manager into commands Frank's POD can understand, and messages from Frank's POD into messages understood by the POD manager and other PODs on the network.

If these same patterns and development models could be applied to the hardware nodes of an embedded device network, software developers could spend more time focusing on the final product, and less time learning how to design within a completely different development paradigm.

Before this can be done however, the degree of parallelism between hardware and software objects as well as what features these patterns require of objects must be determined.

1.3 Prior Work

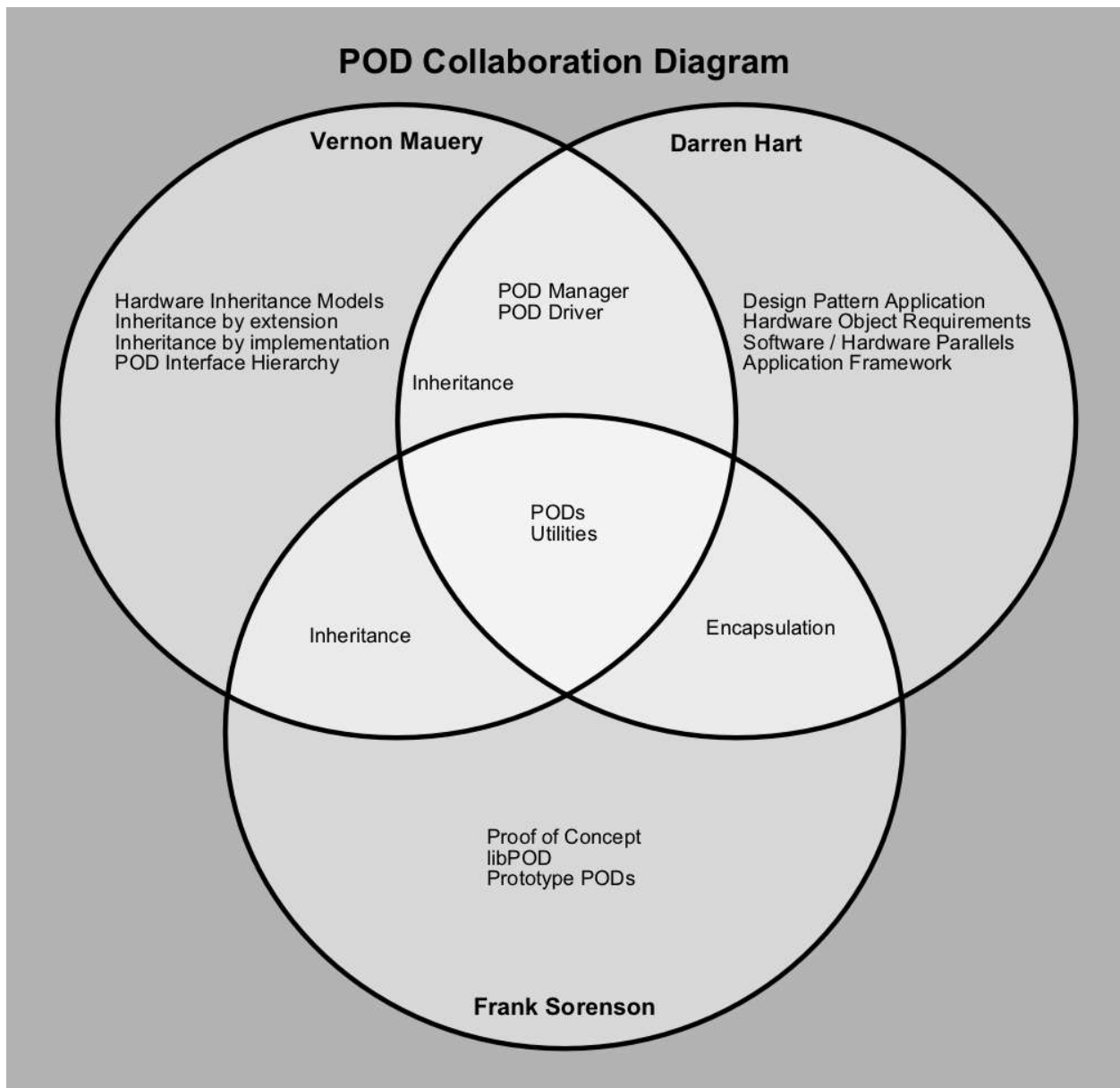
Some work has been done by toy companies to simplify the use of hardware devices. Many motorized construction kits, like those from K'NEX (<http://www.knex.com>), all provide some set of snap together construction pieces (bars, elbows, etc.) that can be combined with motors to create various robotic systems. The systems are generally limited to somewhere between two and six motors and must be manually controlled via a wired control panel with buttons and switches. Lego Mindstorms combine Legos, motors, light sensors, and touch sensors with a microcontroller and an extremely high level programming language to allow hobbyists to design, build, and program unique robotic creations. The individual devices are unaware of each other and communicate only directly with the central microcontroller which only provides for up to three motors and three sensors. [Leg00]

Much effort has been invested in using object-oriented techniques and design patterns to construct applications for control networks, but all the logic and communication is generally handled via large complex software layers. As the cost of microprocessors drops and the speed and computing capacity increases, much of the work done by these software layers can now be pushed down to the devices themselves. These well known design patterns provide a guide of what sort of behavior and functionality should be made part of these devices. [Fil97], [DK96], [Sea98]

Phidgets are a set of hardware user interface devices. Like PODs, they abstract the low level hardware from the end-user, providing user interface (UI) designers with a familiar API. Phidgets are specialized for UI components and have a large number of dependencies, including COM, Active-X, and Visual Basic. [GF01] Our intent is to demonstrate the domain independent applicability of these methodologies, keeping the dependencies to a minimum to reduce the amount of extra research required of the user.

1.4 POD Collaboration

Frank Sorensen and Vernon Mauery are also performing research with PODs. Frank has prototyped several simple PODs as a proof of concept and written a library (libPOD) to pass messages between them. [Sor03] Vernon Mauery is currently working on various inheritance models for hardware objects. Neither has explored the parallelism between hardware and software objects nor have they attempted to determine the facilities that object-oriented design requires of software objects (and therefore hardware objects). The diagram below presents a basic view of the collaboration between these three aspects of POD research.



2 Thesis Statement

By determining the degree of parallelism between hardware and software objects and implementing the necessary features in hardware devices, the design of embedded device networks can be greatly simplified. The designer will then be able to employ familiar software design techniques, treating the hardware devices just like software objects, rather than being required to delve into the methodologies of other domains with which they are less proficient.

3 Methods

Through careful examination of several design patterns, we will determine the characteristics and facilities required of software objects by common object-oriented design techniques. To the extent possible, we will add these facilities to PODs. We will then select several of the patterns most applicable to hardware and use them to implement a POD network. We are currently planning to implement the Singleton, Adapter, Strategy, and Signals and Slots patterns. The following list describes how certain POD demonstrate the applicability of these patterns. The networks we implement will contain examples similar to these.

- Alarm
To demonstrate Singletons and Adapters, we could adapt the interface of a simple alarm and provide it as a singleton object to the POD network. When any device needs to sound an alarm, it simply requests an alarm, the POD manager returns a handle to the only existing one, and the requesting device may then issue commands to the alarm.
- Text / Graphic Displays
We could demonstrate the Strategy pattern by providing a common interface to both the text and graphical displays, the implementation of each will determine how the commands issued by their parent POD are interpreted and displayed.
- Input Panel
By mapping input events to various POD commands, the input panel could serve to demonstrate the applicability of Signals and Slots to networks of embedded hardware devices.

3.1 POD Manager

The POD network will use USB and be managed by a software application called the POD Manager which will communicate with the devices through a Linux device driver. While neither USB nor Linux are essential to this research, their open nature and readily available documentation make them a good choice for academic research.

The POD Manager will work closely with the Linux kernel, providing the facilities needed by the POD network which could not be incorporated into the devices themselves. Since the POD manager is separate from the kernel driver, it will be possible to prototype PODs completely in software. Software PODs will be able to communicate with the POD manager in the same way as hardware PODs. The POD manager along with the kernel driver abstract the hardware aspect of the PODs from the user.

3.2 Linux POD Kernel Driver

The role of the Linux kernel driver is to provide a means of communication with each POD on the network through a set of ioctl calls. The driver provides only basic communication features, such as open, read, write, and close. All policies will be implemented in the POD Manager. This reduces the amount of code running in kernel space, and therefore the potential for disaster.

3.3 Verification

By implementing an embedded device network using common object-oriented design techniques, we will demonstrate the feasibility of treating hardware devices as software objects. This will also show that some minimum degree of parallelism exists between hardware devices and software objects and can be implemented (at least in part) in the hardware devices themselves.

4 Contribution to Computer Science

By applying familiar design methodologies to hardware devices, after determining the requirements of object-oriented design with patterns on these devices, computer scientists will be able to more readily use embedded hardware devices in a variety of situations, from research to rapid prototyping.

5 Delimitations of the Thesis

- This research will not include the development of a production quality simulation environment for PODs.
- We will not address abstracting the bus (USB, I2C, or otherwise) and protocol for PODs to allow the POD manager to communicate over different network types.
- We will not address real-time handling of POD events.
- We will not discuss POD network topologies, bandwidth, or QoS.
- We will not attempt to manufacture production level PODs.
- We will not explore inheritance models in depth.

6 Thesis Outline

Chapter 1 - Introduction and motivation

Chapter 2 - Related work

- A. Motorized toys (K'nex, Lego Mindstorms)
- B. Embedded control networks
- C. Phidgets
- D. Frank's thesis
- E. Vernon's thesis

Chapter 3 - Foundational material (terms, definitions, etc.)

Chapter 4 - Object-oriented parallels between software and hardware

- A. The pointer problem (single physical handle to a device vs. multiple references in software)
- B. Public/private members (host PODs vs. hubs)
- C. Inheritance and composition models (brief overview)

Chapter 5 - Design patterns (if N/A to PODs, explain why here)

- A. Behavioral
 - Mediator
 - Signals and slots
 - Strategy
- B. Creational
 - Singleton
- C. Structural
 - Adapter (object vs. class, hardware or software)
 - Decorator
 - Facade
- D. Requirement on underlying objects

Chapter 6 - Innovations (algorithms, theorems, models, etc.)

- A. Atmel Prototype Board
- B. Linux Kernel Driver
- C. POD Manager
- D. PODs implemented
 - Adapter
 - Alarm
 - Graphic Display
 - Input Panel
 - Text Display
- E. Design patterns implemented
 - Adapter
 - Signals and Slots
 - Singleton
 - Strategy

Chapter 7 - Conclusions and Future Work

- A. Requirements of object-oriented design with patterns on objects
- B. Applicability of design patterns to embedded device networks
- C. Applicable fields
 - Rapid prototyping
 - Simulation environment

D.Future Work

- POD simulation environment
- Support multiple buses and protocols
- Real-time handling of POD events
- POD network topologies, bandwidth, or QoS
- Miniaturization

Appendices - Contains extended results, code fragments, documentation, etc.

7 Thesis Schedule

- Continue literature search (March 2003)
- Setup the POD development environment (atmel tools, programming equipment, etc.) (March)
- Define general POD interface and interPOD communication (working closely with Frank Sorenson) . Write a kernel driver to support the POD interface (April – May)
- Select a few PODs to build (VFD/debugger, motion table, etc.) (May)
- Design the interface to the selected PODs (May – June)
- Write the AVR software for those PODs (June – July)
- Explore various ways in which software design patterns map (or don't map) to the POD network (August – September)
- Write thesis (August – October)
- Defend Thesis (December)

8 Bibliography

- [CIM98] Compaq, Intel, Microsoft, NEC. *Universal Serial Bus Specification Revision 1.1*. <http://www.usb.org>, 1998.
- [DK96] Per Dagermo, Jonas Knutsson. *Development of an Object-Oriented Framework for Vessel Control Systems, ESPRIT III/ESSI/DOVER*. Dover Consortium, 1996. An overview of using design patterns to implement an extensible framework for vessel control systems.
- [Eck01A] Bruce Eckel. *Thinking in Patterns with Java: Problem-Solving Techniques Using Java*. MindView, Inc., 2001. Discussion of design patterns, the theory, their purpose and application.
- [Eck00] Bruce Eckel. *Thinking in C++, 2nd ed. Volume 1*. MindView, Inc., 2000. A practical to guide beginning to think in an object-oriented language, specifically C++.
- [Eck01B] Bruce Eckel. *Thinking in C++, 2nd ed. Volume 2*. MindView, Inc., 2001. A application oriented guide to thinking in object-oriented ways, discussing more advanced topics such as containers and design patterns.
- [Fil97] Jose M. Filgueira. *A Distributed Object-Oriented Telescope Control System Based on RT-CORBA and ATM*. International Conference on Accelerator and Large Experimental Physics Control Systems, 1997. High level discussion of the various software layers involved in the telescope control system.
- [Fli00] Detlef Fliegl. *Programming Guide for Linux USB Device Drivers*. <http://usb.cs.tum.edu>, May 19, 2003. A detailed overview of USB devices and the Linux USB subsystem API.
- [GF01] Greenberg, S. and Fitchett, C. *Phidgets: Easy development of physical interfaces through physical widgets*. Proceedings of the ACM UIST 2001 Symposium on User Interface Software and Technology, November 11-14, Orlando, Florida. ACM Press, 2001. An overview of the theory behind phidgets (physical widgets), their architecture, and application.
- [Gre03] Douglas Gregor. *Boost Signals Documentation*. <http://localhost/doc/libboost-doc/HTML/doc/html/signals.html>, May 20, 2003. A description of signals and slots in respect the boost implementation by Douglas Gregor, including the API and a tutorial.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. Comprehensive discussion of creational, structural, and behavioral software design patterns.
- [Leg00] The Lego Group. *Lego Mindstorms, Robotic Invention System, System 2.0*. The Lego Group, 2000. An overview of the Lego Mindstorms system and usage, with example designs and programs.

- [RC01] Alessandro Rubini, Jonathan Corbet. *Linux Device Drivers, 2nd Edition*. O'Reilly, 2001.
Comprehensive guide to writing kernel device drivers, updated for version 2.4.x of the Linux kernel.
- [Sea98] Seagate Technology, Inc. *Object-Oriented Devices: Description of Requirements*. Seagate Technology, Inc., 1998.
A description of the requirements for devices to act as objects in an object-oriented networked attached storage system.
- [Sor03] Frank Sorenson. *PODs: Pel Object Devices DRAFT*. Brigham Young University, 2003.
An as yet unpublished thesis dealing with the basic implementation of several PODs.

9 Artifacts

- Example PODs (Adapter, Alarm, Graphic Display, Input Panel, Text Display)
- Linux POD kernel driver
- POD Manager

10 Signatures

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis proposal submitted by

Darren Hart

This thesis proposal has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date

Kelly Flanagan

Date

Dan Olsen

Date

Mike Jones