INHERITANCE MODELS IN OBJECT-ORIENTED HARDWARE USING PHYSICAL OBJECT DEVICES

by

Vernon H. Mauery

A thesis 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

April 2004

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Vernon H. Mauery

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

Date	J. Kelly Flanagan, Chair
Date	Dan R. Olsen
Date	Dan Ventura

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Vernon H. Mauery in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

J. Kelly Flanagan
Chair, Graduate Committee

Accepted for the Department

David W. Embley
Graduate Coordinator

Accepted for the College

G. Rex Bryce, Associate Dean
College of Physical and Mathematical Sciences

ABSTRACT

INHERITANCE MODELS IN OBJECT-ORIENTED HARDWARE USING PHYSICAL OBJECT DEVICES

Vernon H. Mauery

Department of Computer Science

Master of Science

Several companies have attempted various ways of designing hardware devices that are simple to interface with and use. The toy industry wants to make electronic toys that kids and young adults can use, while do-it-yourself companies target the home electronics enthusiast who wants to interface with his electronic gadgets. The result has been classified as object-oriented hardware. This hardware offers features such as encapsulation, abstraction and inter-device communication, but it has not yet employed the concept of inheritance – one of the most powerful elements of software objects. Physical Object Devices (PODs) have a clean architecture that will allow us to easily extend the objects and find the requirements that inheritance places on its objects and environment.

By looking for parallels in hardware and software objects, we can employ inheritance models in hardware objects. These parallels will define the requirements for an

architecture using Physical Object Devices that has many of the same properties as software objects – including inheritance. Inheritance leads to the rapid development of more complex systems because object extension and interface inheritance offer code reuse and common interfaces. In addition to these basic time and money saving benefits, hardware inheritance will offer many of the properties that allow programmers to create high quality designs in software, such as encapsulation, member protection, and polymorphism. Developers who use inheritance with PODs will be able to create cleaner designs and more complex systems in less time and with less duplication of code.

ACKNOWLEDGMENTS

I would like to thank all the people who have helped me in my research. Thanks to: Kelly Flanagan for his support, advice, and financial assistance; Dan Olsen for his ideas and endless questions that made me think; Dan Ventura for his advice; Darren Hart, Myles Watson and the rest of the Performance Evaluation Lab for their help; and last but not least, my wife, Lauren, and my daughter, Nicole, for being so patient with me while I spent so much time at school.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Background

Hardware Is Still Hard

In recent history, we have seen a division in the computing world. On one side, we are seeing faster, better desktop computers, and more and more powerful servers. On the other side, we see embedded devices that are becoming increasingly ubiquitous [1]. We see networks of microcontrollers in cars, houses, assembly lines and many other places [2]. Our ability to create "smart devices" is driving the market such that people are expecting more and more embedded devices everywhere. This demand is putting pressure on computer scientists and electrical engineers to design these devices with ever-increasing speed and complexity.

To combat the complexity of embedded development, more options are becoming available. Several groups have done research to find ways to make embedded development simpler. This area of research is called *object-oriented hardware*. The idea is to abstract some of the programming and circuit design away from the developer to make more complex designs easier to create. Physical Object Devices (PODs) are one example [3]. They offer abstraction from the hardware and essentially allow the users to treat hardware devices as software objects.

Benefits of Abstraction

Abstraction in computing has endless levels – from virtual memory down through the ranks of the operating systems, and then even more layers of abstraction in the hardware. Abstraction offers a different, usually simpler, view of a complex system, exposing a usable interface to the user [4]. In the realm of hardware, abstraction

is essential in order to make any progress. Consider programming a computer not by using a high-level language, but rather by individually manipulating the CMOS transistors to work as logic gates and state machines. Understanding a computer on these terms seems impossibly difficult, but using the abstraction that a CPU offers, we suddenly have memory, instructions, registers and I/O devices. This is a much more usable view of a computer. Similarly, in embedded development, we can find similar ways to abstract the embedded hardware. We can view devices as software objects that have a known interface. Then we can use familiar programming techniques to develop and create more complex embedded systems.

1.2 Hardware Abstraction

In the realm of toys, we see a trend toward high-tech toys that are reconfigurable by the kids that play with them. In this area, kids want to be able to assemble the parts and then control them either by programming them or using some sort of remote control. The idea of object-oriented hardware is very useful in this area because it offers an abstraction from the hardware that kids and young adults can easily understand. Then, without needing to know how to wire up a motor or interface a sensor, they can create a device that crawls toward light.

Lego MindStorms offer this level of abstraction [5]. However, they lack the other elements of object-oriented design. The individual devices have no means of communicating directly with each other and have no knowledge of each other. All communication and control is done by an onboard RCX computer. But the encapsulation and abstraction MindStorms offer are very good. All the parts snap together to make simple electrical connections. The interface to the sensors and motors is essentially part of the programming language; it is almost entirely visual, using icons and flow charts to create program control. This solution is so simple, MindStorms are designed for ages 12 and up.

OOPic (Object-Oriented Programmable Integrated Circuit) is another set of devices that attempt to allow object-oriented programming with hardware [6]. They are able to network together using I2C and access functions that wrap hardware. However, the only facet of object-oriented programming they offer is encapsulation. They encapsulate all the hardware functions into an object and abstract the details away from the user. This is useful, but again, it lacks some of the more powerful constructs of object-oriented design.

Phidgets are another available hardware/software interface [7]. Much like other object-oriented hardware, Phidgets provide a software-programmable hardware interface. Most of their work has been in applying the ideas of a GUI to hardware. One application listed on the Phidgets website is a flight simulator that uses Phidgets as the controls for the pilot. Writing an application that uses Phidgets is simple, but uses some heavy duty, system dependent software, namely Visual Basic and COM. This is fine for many applications, but too much overhead for many embedded applications.

1.3 Physical Object Devices

Object-oriented hardware is a relatively new paradigm. Some research has been done in this area to add abstraction and simplify working with hardware. Physical Object Devices [3] are probably the most advanced hardware abstraction architecture that we have to date. PODs give hardware objects a common interface and the ability to interact and connect without a soldering iron. They allow the user to treat the hardware much like a software object in the sense that PODs accept commands and queries and perform tasks on user data or data from their environment. PODs currently are not capable of inheritance, which is one of the facets of object-oriented design that offers a great deal of power.

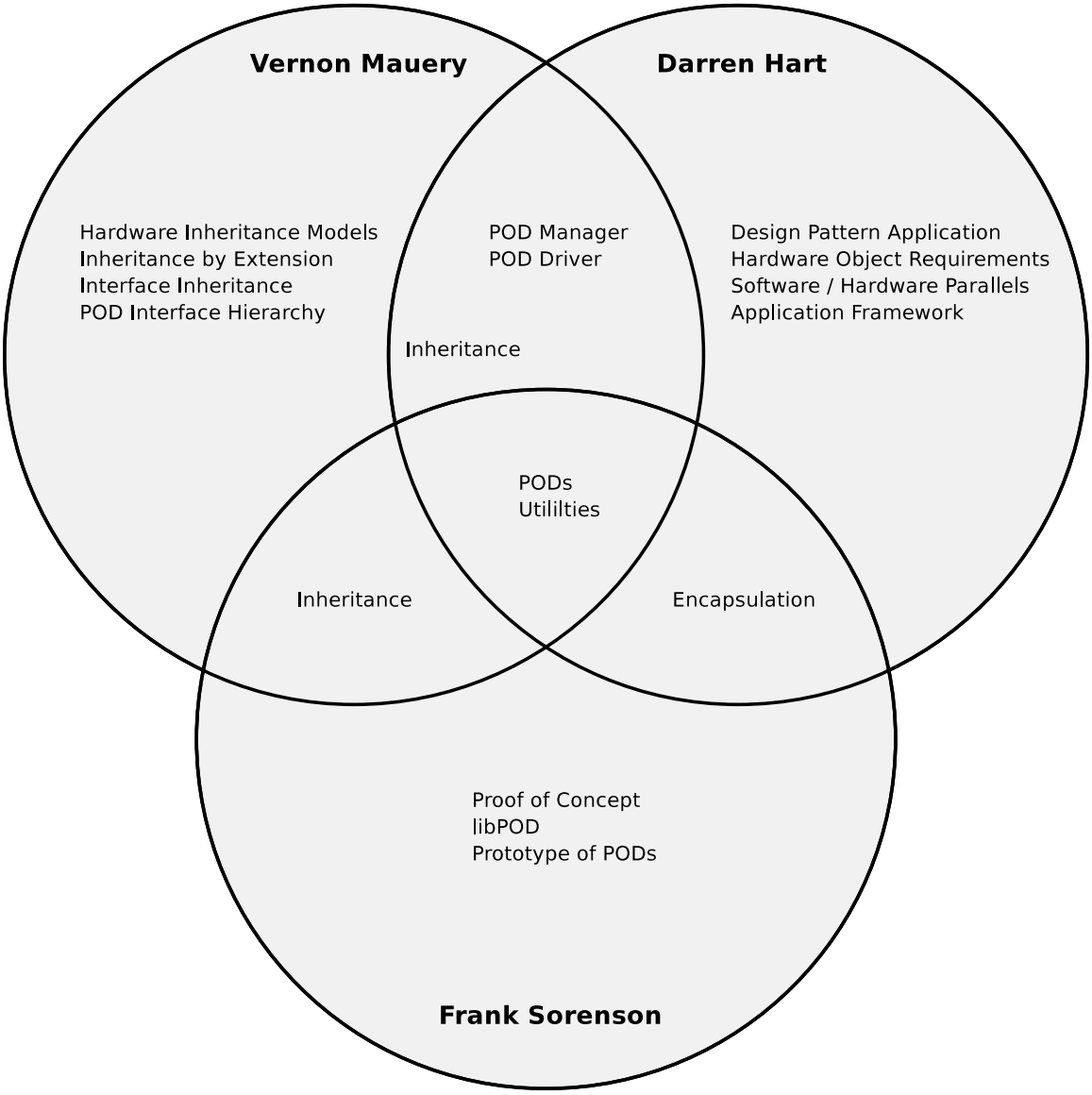

Figure 1.1: POD Collaboration Diagram

POD Collaboration

PODs started in the Performance Evaluation Laboratory at Brigham Young University's Computer Science Department. Frank Sorenson started off the research with a proof of concept and prototyping of PODs [3]. His research was a springboard for the further research being done at the moment by the author and Darren Hart [8]. Some collaboration has been done in areas that were covered by multiple research areas. Figure 1.1 diagrams the collaboration between these three areas of POD research.

Finding Parallels between Software and Hardware

Current object-oriented hardware, advanced as it is, still lacks some basic object-oriented design capabilities. So far, encapsulation, abstraction, and data protection are the only object-oriented principles designers have taken advantage of, yet these can be done in structural programming just as well. The object-oriented paradigm has much more to offer than just these three concepts. Inheritance is one of the most powerful object-oriented constructs, but although structural programming strives to provide this functionality, it has not yet succeeded. Inheritance in hardware, like software, can be thought of in two ways: extending implementations and implementing interfaces. In Java, we would use the keywords *extends* and *implements* respectively to refer to these types of inheritance.

Computer scientists already know and understand the benefits of object-oriented programming in software. Because we are applying software ideas to hardware, we can get some of the same benefits – and one of the most important of these is inheritance. Inheritance leads to the rapid development of more complex systems because object extension and implementation offer code reuse and common interfaces. When developers create more complex systems, they can use object-oriented hardware to make larger systems without having to completely redesign when adding or changing

a subsystem; if a subsystem breaks, fixing that part or replacing it with one of the same interface will be trivial.

Since the 1950s when electronic computing made its debut [9], computer scientists and computer engineers have been looking for ways to make programming easier. Since then, they have come up with thousands of programming paradigms and about as many programming languages. Of all these creations, some have stayed around while others have been forgotten. Each language and paradigm had at least one thing in common: abstraction in the name of simplicity. Because human programmers think in more abstract terms than machine code, it is only natural for us to find some way to translate our abstract ideas into the tedium of bits that computers can use. High level languages in general have done this, while object-oriented languages offer even more abstraction to the way we think.

The recent focus on rapid development and code reuse [10] sheds a good light on object-oriented programming. Object-oriented programming offers some abstractions that make rapid development and code reuse much simpler. Specifically, inheritance and composite models both show promise because of abstraction, modularity and code reuse [11].

Thesis Statement

The current state of object-oriented hardware lacks some of the features, such as inheritance, that make software objects so useful and powerful. Finding parallels in software and hardware development will enable computer scientists to implement the prevalent software inheritance models in hardware objects, thus allowing for more complex POD systems and cleaner POD designs.

Thesis Layout

The remainder of this thesis is outlined as follows: this chapter touched on some of the work that has been done in the area of object-oriented hardware and gives some

background on the principles of object-oriented design that are used with inheritance in PODs. Chapter 2 explores the inheritance requirements of PODs. Chapters 3 and 4 deal with how the POD system meets the requirements for inheritance. Chapter 5 explains the experimentation and analysis of two POD systems. And finally, chapter 6 summarizes the requirements for POD inheritance, emphasizes some of the benefits of inheritance and lists some areas for future research.

2. INHERITANCE MODELS IN HARDWARE

A good starting place for determining the properties that our hardware inheritance models exhibit is to start by studying the familiar models of inheritance in software. From this, we can make some generalizations that can lead us to find what is expected by an object that can inherit from another.

2.1 Inheritance in Software

The inheritance model comes naturally as a part of object-oriented programming and is a very powerful abstraction mechanism. By grouping similar classes together, we can define a base class, or superclass, that embodies the common attributes of the group. Then, each individual class can be defined by specifying the differences between the base class and the new subclass [4]. In other words, we can say "a thingy is like a foo but it has a bar too." Explaining in differences is a natural way for us to think because we have been doing it since birth and it conveys so much information in so little speech. For example, explaining what a zebra is like to a child can most simply be done by saying that a zebra is like a horse except that it has black and white stripes. From this, a child will generally assume that zebras are four-legged horse-shaped animals that like to eat grass, but their stripes differentiate them from horses [11]. The rest of the section will examine the relationships between the classes shown in figure 2.1. This figure shows a class hierarchy that includes many of the properties that will be discussed.

In terms of the zebra example, we could call the horse a superclass and call the zebra a subclass. A superclass is the class that contains the common features of a group of similar classes. Then, each subclass derives from it to create something slightly

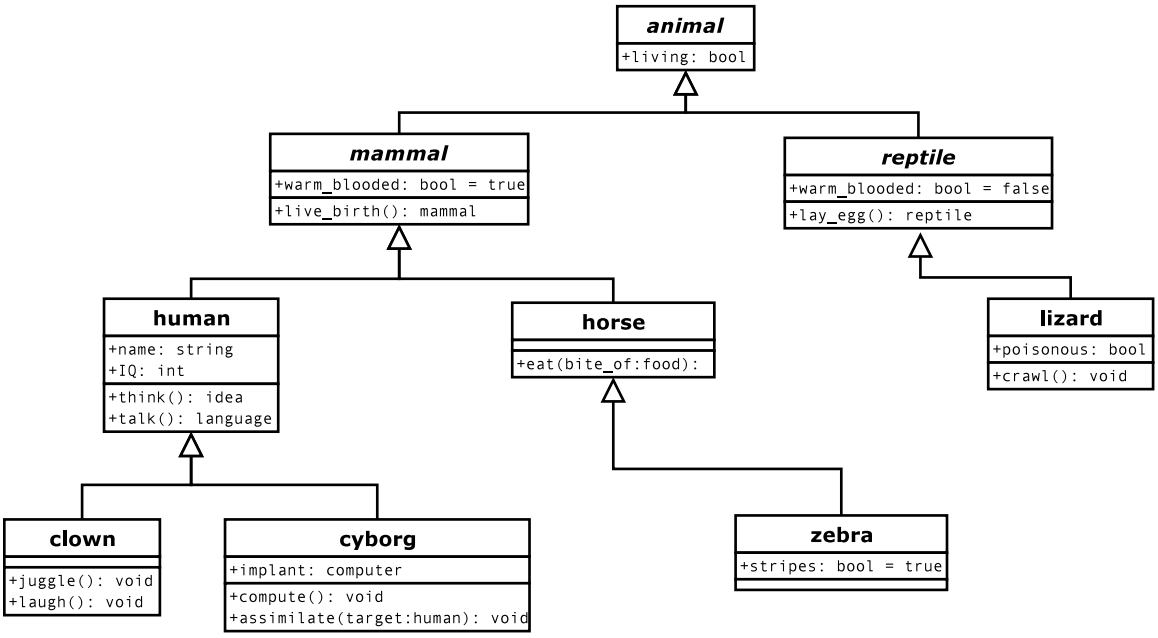

Figure 2.1: Inheritance Example

---

different. Figure 2.1 has many classes, some of which are subclasses, superclasses or both. In the latter case, the class both inherits some properties and then passes those properties and its properties on to its subclasses. Sometimes we need to be so abstract in describing a class that it is no longer even possible for that object to exist. For example, a human and a horse are both mammals because they are warm-blooded and give live birth. But can we ever find an animal that is only a mammal, not some subclass of a mammal? No, because a mammal is an abstract class. The abstract classes in the figure are animal, mammal and reptile. Abstract classes are useful to programmers when they need to create an interface. In programming, an interface forces all derivatives to implement a given set of routines in order to be able to treat all instances the same.

In addition to the advantages that we get from the information transfer of inheritance, we can exploit the differences between superclass and subclass using polymorphism. Polymorphism is a word made of Greek roots that means “many forms.”

Applying polymorphism to programming, creates virtual functions and overriding methods. The many forms part comes to play when a subclass overrides an implementation of a superclass method. Then, although the subclass and superclass have similar interfaces (at least as far as the overridden method is concerned) they have different forms, meaning they behave differently.

Extension Model

Extension is a simple form of inheritance in which the superclass is concrete, or non-abstract. This means that the subclass can reuse code from its superclass. To make it useful, we find a class that is close to what we want to make and then specify the differences. In some cases the difference may be that the subclass requires a few more methods that extend its API. In other cases the difference may be the subclass requires a new member that adds some new functionality. In either case, we extend the superclass to make a more specialized class that can do more or do something better.

For example, we start with a human as a superclass. To extend the human, we could teach it to juggle, tumble and laugh (adding methods) and we have a clown. The clown class is a specialized subclass of human because it can do all the things a human can do plus the added tricks we taught it. Now, we could also start with the human and extend it by implanting a computer in its brain. This new subclass of human is a cyborg. A cyborg can still do the things a human can do but it is once again an extended, specialized subclass.

Interface Inheritance

Interface inheritance is another application of inheritance to simplify programming. The idea is to create a hierarchy of abstract classes that provide an interface to implement. The advantage of creating interfaces and implementing them separately is the possibility of different implementations. In software this may simply mean

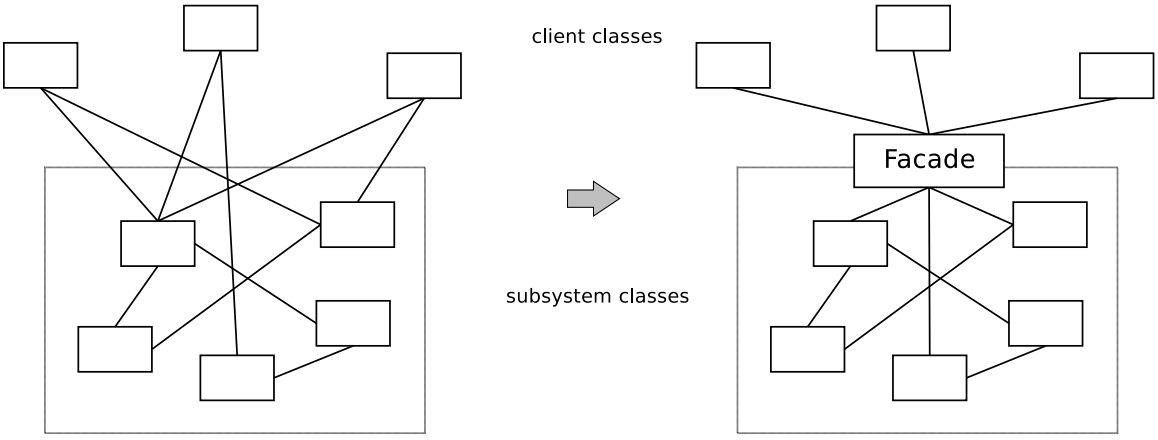

Figure 2.2: Composite Example

---

a faster or smaller way of writing the class. This allows a way to make "drop-in" replacements for objects and emphasizes the modularity of a class.

For example, if you need to use a foo, you could buy one from CheapFoos.com or from your local hardware store. Because every foo must implement the foo interface, you are safe to get your foo from anywhere you want. Each foo may have a different implementation, but they should all be compatible at the interface level.

Composite Model

Sometimes it is easier to describe classes in terms of their parts rather than in terms of their interface. When this happens, it is often easier to make a new class that is a collection of smaller classes that acts as a wrapper to bind the classes together and supervise interactions. This model is known as the composite model because rather than inheriting attributes from superclasses, we collect components and put them together into one larger, more useful class.

There are several views that one can take when using the composite model. A strictly composite model could be one that simply acts as a container for several objects while implementing a minimum interface that allows external interaction with the objects. A common object-oriented design pattern called Facade [12] is used to

collect a group of related classes together and expose a unified API that simplifies the use of the classes. As shown in figure 2.2, the Facade acts as an intermediary between the client classes and the subsystem classes. Facades are unique because they work by delegation. The Facade itself does not need to do much work besides figuring out which part of the system should handle the request it just received.

Some would argue that inheritance that includes adding more classes and wrapping them is also a form of composition. This is a very fine line to argue over because it exhibits properties of both the inheritance model and the composite model. When we speak in terms of "is a" and "has a", this problem clearly is part of both models. The new class is a subclass of the original class and has some components as part of it. Because this form is a mix of both inheritance and composition, we will treat it as a form of extension inheritance for this work.

2.2 Inheritance in Hardware

The goal of being able to treat hardware objects like software objects by exploiting parallels leads us to explore the ways that we could think about inheritance models in hardware. We can think of inheritance in hardware in the same three categories as we did in software. By defining what extension, interface inheritance and composition in hardware mean, we can then determine what properties the hardware would need to exhibit in order to implement each model.

There are, however, some obvious differences between hardware and software objects that need to be reckoned with. Each hardware device is in a different address space because they are all running on separate microprocessors. This makes it more difficult to access superclass members but trivial to restrict access to protected and private members. Hardware cannot be replicated in the same way that an object in memory can be copied. This means that we cannot pass hardware objects by value, only by reference or by pointer.

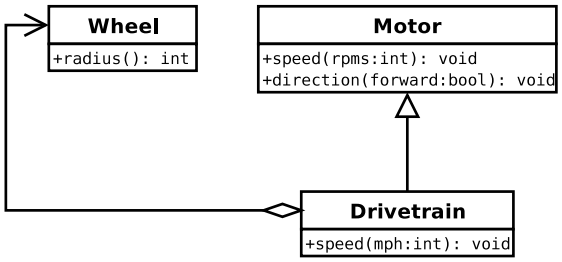

Figure 2.3: Hardware Extension Example

Extension Model in Hardware

The extension model in hardware follows the same line of thinking that extension in software does. We start with a base class that contains all the common features of a group of classes. Because we are often adding another piece of hardware to extend, extension and composition in hardware are not much different. This base class can then be extended by adding more member classes and methods to make a new subclass. The difference is that in hardware extension, we are adding hardware classes to extend a hardware superclass. Extension that adds only methods or non-aggregate typed variables is trivial and will not be dealt with in this discussion.

To demonstrate the idea of hardware extension, we will examine a simple example. As shown in figure 2.3, we start with a Motor class and extend it by adding a Wheel. A Motor has the properties of direction and speed in RPMs with member methods that can retrieve or modify these properties. A Wheel has the property of diameter and a method to retrieve it. If we extend the Motor with the Wheel, we then have a new device we can call a Drivetrain. This new subclass of Motor can now also get and set the speed in m/s rather than only RPMs because it now has a notion of distance related to rotations. But we can still treat it as a Motor if we need to because it extends the Motor class. We can think of this relationship in terms like “is a” and “has a”: the Drivetrain is a Motor, and has a Wheel.

The benefits that this model provides are very similar to the benefits of extension in software. Virtual functions, which are one of the biggest benefits polymorphic inheritance offers, can also be available in this hardware model. Member access protection comes naturally as a result of the objects working in different address spaces. All access to member variables must be done through member methods and only the public member methods would have to be exported. We can also take advantage of code reuse because the subclass still has access to the superclass's methods. An example of this is that the Drivetrain will still need to set the speed of the motor and does not have to rewrite the code that does that, but can simply call the superclass's `speed` method.

This form of inheritance places some requirements upon the architecture. Somehow, the Drivetrain object needs to acquire a handle to the Wheel that is connected to it in order to operate correctly. If there is no Wheel available or some other error occurs, the Drivetrain needs to be able to handle and throw exceptions. In order for the Drivetrain to request the size of the Wheel, the two objects would need to be able to communicate. There needs to be some way of offering virtual binding for methods that get invoked on the objects. Access to superclass methods must be available for virtual methods that still need to use methods defined by the superclass.

Interface Inheritance in Hardware

Interface inheritance is the idea of dividing classes into a hierarchy by function. At the top would be a generic object class that exports the interface that every object must implement. Below that in the hierarchy, we might find subclasses such as input, which derives from the base object, and sensor, which derives from input. At each level, the classes become more specific and add to the interface a new set of methods that objects of this class must implement. This means that an object that is a sensor must implement the sensor interface, in addition to the input and

base object interfaces. This sort of hierarchy offers a very useful backward or upward compatibility that allows users to use objects by making calls that are part of the inherited interface.

While interface inheritance does not offer any of the benefits that polymorphic inheritance does, such as virtual methods or information hiding, it has a few features of its own. Interfaces define access to the class, and possibly behavior of the class, but not implementation. This opens up possibilities for manufacturers and designers because with the standard interfaces set, anyone could create an object that implements the interface using any proprietary code or new technology they desired. Then, this object could be replaced by another implementation if a user wanted.

Because objects may implement more than one interface, some mechanism that allows this must be in place. Each object would also be required to implement all the classes up the tree to the base object in order to gain the full benefit of interface inheritance.

2.3 Theoretical Requirements of Inheritance

Inheritance, as defined in this chapter, has very close parallels in software and hardware. While the implementation of inheritance in each realm may be very different, we can create behavioral inheritance models in hardware that mimic the patterns found in software. But in order to do so, we must meet a list of requirements that will allow both models to behave similarly. After studying the generally accepted properties of inheritance in software, looking for parallels in hardware and determining the characteristics of the hardware systems that could be built, we have come up with the following list of requirements that hardware objects must implement in order to behave in terms of inheritance as software objects do:

- Construction and destruction
- Virtual methods

- Access to superclass methods
- Handle acquisition
- Object-to-object communication
- Exceptions (handling and throwing)
- Multiple interfaces

Using these features, developers could create a system that exhibits behavior in hardware that is very similar to inheritance in software. There may be other requirements that this theoretical study did not expose, so to be sure that we have a comprehensive list before implementing our hardware system, we will take a look into some practical system designs in chapter 3.

3. POD INHERITANCE REQUIREMENTS

3.1 Design Example Exploration

After taking a look at the theoretical characteristics of inheritance, we need to examine the practical uses of inheritance. This means taking a look at several design examples to see what kind of requirements they impose upon a system. To get a broad range of input, we will look at ten examples that cover three general areas of extension: interface inheritance, extension, and composition. These examples will give us a good idea of what requirements inheritance places upon PODs. In addition to finding the requirements of inheritance, we will also contrast the inheritance model construction with construction of a similar system without using inheritance.

Extension Examples

First we will look at the examples that are done purely through extension. Of the ten, we have four that fit this group: Anti-lock brake system (ABS), Automatic_door_light, Drivetrain, and Smart microphone. The ABS does a very good job of illustrating the properties of this group, so we will examine it in more detail than the other three. We will look at what it takes to construct such a device and then explore the advantages this mode of construction has over construction of a similar device that does not use inheritance.

An ABS is a device that uses a feedback system to determine whether or not to assert the brake. Normal behavior could be defined as follows: the driver presses the brake pedal; the ABS tests wheel speed to see if it is greater than zero; if not, apply brake and quit algorithm; otherwise, apply brake and monitor wheel speed; if wheel speed drops to 0, release brake for a short time and then reapply it; continue to loop

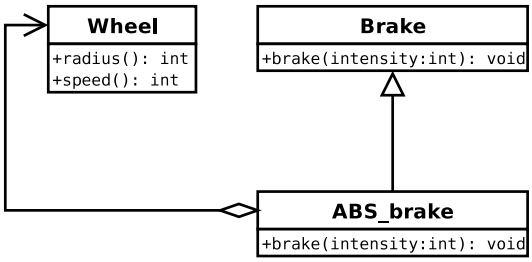

Figure 3.1: ABS Brake Hierarchy

---

in monitor section while brake pedal is pressed. Another important thing we need to consider is that this system must be a drop-in replacement for non-ABS brakes. As we can see, in this algorithm, there is an interaction between several pieces of hardware. We can break the hardware into the separate units of brake pedal, brake actuator and wheel.

The object-oriented approach would be to create an ABS_brake class (as shown in figure 3.1) that has the same interface as the standard non-ABS Brake class and use the existing Brake_pedal class. Then, deployment of these ABS_brakes to replace the non-ABS Brakes would be trivial. From what we have seen up to this point, inheritance looks like the prime solution. If we could extend the standard Brake class by giving it access to a Wheel and some intelligence, we could create a new class that exhibits the correct behavior. The derived class, ABS_brake, shown in figure 3.1, is the class we would like to build. The most obvious use of virtual functions is with the `brake` method, where ABS_brake overrides the Brake class to add the new behavior before actually applying the brake. Extending the Brake class would give us an API similar to the one show in table 3.1. We see that most of the methods that Brake provides are overridden by the ABS_brake class.

In order to actually implement this, we need a way to override a method but still be able to access the superclass method. Then, at runtime, the binding could be

Method Name	Fulfills API	Overrides Superclass	Comments
ABS_brake	POD	n/a	Constructor
~ABS_brake	POD	n/a	Destructor
name	POD	yes	returns "ABS_brake"
url	POD	yes	returns URL where more info can be found
set	output_pod	yes	synonym for brake
get	output_pod	yes	returns braking intensity
ready	output_pod	no	returns braking status
go	actuator_pod	yes	synonym for brake
brake	brake_pod	n/a	applies brake according to ABS behavior

Table 3.1: ABS Brake Interface

determined in a similar way to dynamic binding in software. Extending in hardware would mean that we need to modify and extend the software that resides on the device we are extending to make the device aware of the extended API and functionality. While an amateur may not want to do this, there are ways to make this painless and simple. After all, the hardware and circuitry is all present and working, so extension is mostly a matter of defining the polymorphic methods and the interaction with the new member POD. This also means that from the POD constructor, we need to be able to request a POD handle. For example, the ABS_brake needs a Wheel, so in its constructor, it could request the handle of the Wheel it is mechanically connected to. Thus PODs not only need to be able to request handles to other PODS, but they need to be able to request handles to specific PODs. Because faults can happen at construction time (such as no POD available) the PODs need to be able to handle exceptions and throw them depending on circumstances.

After adding the ABS_brake code to the Brake device, it now knows how to behave as an ABS_brake if it is constructed that way. Upon construction, the ABS_brake

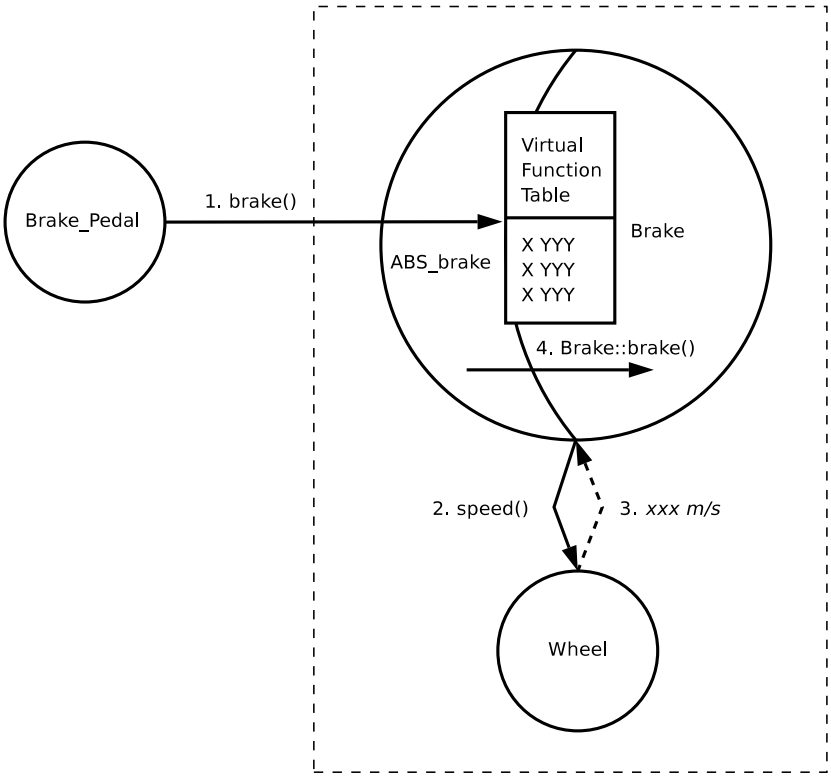

Figure 3.2: ABS Brake Interaction

requests the connected Wheel handle by serial number from the POD manager. If that Wheel is not available, it throws an exception. Otherwise it stores the handle for later use. The application software then can register the `on_pressed` callback to call the ABS_brake's `brake` method. As shown in figure 3.2, the Brake_pedal calls the `brake` method. This then gets looked up in the virtual function table and determines that it will run the `ABS_brake::brake` method. This method calls the `speed` method on its Wheel handle to determine the current wheel speed. Then, it makes a decision on whether or not to actually assert the brake. If the wheels are not locked, the `Brake::brake` method is called, which asserts the braking mechanism.

A procedural approach might be to use the existing three classes (Brake_pedal, Brake, and Wheel), and create an application that interacts with them as a controller. This application would need some way of notification for when the Brake_pedal was pressed. Then, when it received this notification, it would need to query the Wheel for speed and use this information to determine whether or not to apply the Brake. This could be done with the current object-oriented hardware technology, but that solution fails to capture the elegance that inheritance offers in this case. By pushing the intelligence down to the lowest layer possible, we can mimic the software world more closely, thus making hardware objects easier to use with a familiarity factor. Just as software procedural methods fail to capture true encapsulation, this model in hardware also requires the application to contain some of the runtime code that determines the interaction between the three objects.

The other three examples we worked with in this area were the Automatic_door_light, the Drivetrain and the Smart_microphone. We defined a Drivetrain to be an extension of the Motor class with a Wheel, which extended the interface of the Motor by adding the ability to set the speed in m/s or MPH rather than just RPMs. The Smart_microphone is an extension of the Microphone class with a digital signal pro-

cessor that has the ability to filter the incoming sounds according to the user-defined digital filter. Finally, the Automatic_door_light, which has the same interface as the Light POD, but has some extended behavior, such as turning on if it senses motion or if its timer expires. As shown in table 3.3, these three examples exhibit very similar properties and have similar requirements to the Anti-lock brake system. Because of the similarities, we will not go into a deeper discussion of them here.

Interface Inheritance Examples

Next, we will look at a group of devices that can be categorized as using interface inheritance. In other words, the systems depend on objects to fulfill certain interfaces without requiring them to be specific types of devices. In the study we did, there were three devices that fit into this category: Touchpad_display, Garage_door_opener, and Baby_monitor_system. The Touchpad_display is an effective example to show the details of this group.

We will construct a Touchpad_display out of a Touchpad and a Display. The reason we are calling this interface inheritance is because we want to be able to use any Touchpad and any Display – a Graphical_display should work as well as a Text_display as long as they both implement the Display interface. Another feature of this example is that while it is created as an extension of a Display with a Touchpad, it will implement multiple interfaces and thus be able to be treated as either a Touchpad or a Display. This is shown in table 3.2, where the Touchpad_display implements both the display_pod interface and the touchpad interface.

The behavior of the Touchpad_display will encompass the behaviors of both a Touchpad and a Display. In addition, we can take advantage of some interaction between the two devices. The Touchpad can request the number of rows and columns in the display and divide its touchable area into the same matrix. Then, when it is touched, it can make a call to the display to set the cursor position. All this could

Method Name	Fulfills API	Overrides Superclass	Comments
Touchpad_display	POD	n/a	Constructor
~Touchpad_display	POD	n/a	Destructor
name	POD	yes	returns "Touchpad_display"
url	POD	yes	returns URL where more info can be found
set	output_pod	yes	synonym for put_char
ready	output_pod	no	returns display status
rows	display_pod	no	returns number of rows on display
cols	display_pod	no	returns number of cols on display
cursor	display_pod	no	sets cursor position
clear	display_pod	no	clears display
put_char	display_pod	no	writes a character to the display
put_text	display_pod	no	writes a string to the display
get	input_pod	no	gets the position of the last touch
have_input	input_pod	no	returns whether or not have new input
on_input	input_pod	no	callback to run when have a new touch
sample_rate*	input_pod	no	get/set the sample rate for the touchpad
on_press	touchpad	no	callback to run when touchpad is touched
on_release	touchpad	no	callback to run when touchpad is released
on_move	touchpad	no	callback to run when point of contact moved
height	touchpad	no	returns height of touchpad
width	touchpad	no	returns width of touchpad

Table 3.2: Touchpad_display Interface

happen automatically, without external interaction. Because the two devices are acting together as one, it also simplifies external interaction; the same POD handle (the Touchpad_display handle) can be used to get touch coordinates or set text to the Display.

We note that while this example technically works in the same manner as the ABS_brake example above, programmatically speaking it is very different. This example does not require virtual functions to override the superclass's methods. The previous example did not require interface inheritance. But, there are a lot of common requirements because of the similarities of construction. This example also requires the ability to request and store another POD's handle in the constructor. It also needs to be able to throw and handle exceptions, communicate with other PODs, and have access to the superclass methods. One interesting requirement of this example is that it implements multiple interfaces. While supporting multiple interfaces is not directly a requirement of interface inheritance, it is a nice feature to support for designs like this.

The procedural approach to construction of a similar device is possible in a similar manner as described above for the ABS_brake example. An application would need to be created that acts as the feedback loop from the touchpad to the display. In addition to control, it would need to create some sort of proxy that could act as the interface to the two devices. Once again, this is all possible, but by pushing the intelligence down as close to the hardware as possible, we are able to get more software-like behavior from the PODs, which makes them easier to use.

The other two examples we worked with in this area were the Garage_door_opener and the Baby_monitor_system. We defined a Garage_door_opener to extend the Motor class with some sensors to determine when to open and close the door. The Baby_monitor_system is essentially an application that uses an array of sensors and

an alarm. In order to use any kind of sensor, it depends on interface inheritance for the sensors. As shown in table 3.3, they exhibit very similar properties and have similar requirements to the Touchpad_display. Because of the similarities, we will not go into a deeper discussion of them here.

Composition Examples

Finally, we look into some devices constructed using composition. We can construct these devices by choosing the POD that is most similar to what we want. To perfectly parallel software, composition should be done by creating a new POD that contains all the POD pieces that it needs and then export an interface that makes sense according to the pieces while acting as a delegator. This is exactly what the composition diagram showed in figure 2.2. Since the extra layer of communication that the delegator POD adds is unnecessary, we propose two methods of getting around this. First, we could pick one object of the composition and extend it by adding the other objects. While this is not completely accurate as far as software parallels go, it is a simple solution to the problem. The second solution would be to use a software application that acts as the delegator class. This would probably be the simplest method of doing composition that does not exhibit inheritance properties.

But to show the simplicity that the extension model offers even this category of problems, we will walk through the Robot example. Once we come up with a definition of the device in terms of "is a" and "has a" relationships, determining the inheritance properties that it uses is relatively simple. We chose to think of a Robot as a Drivetrain that has some sensors. We also wanted the Robot to have some internal behavior, such as making motion decisions based on the sensor data that it collects. This could be done using the sensors' callback methods or through polling.

The three devices in this category that we examined are the Robot, the Calculator, and the Door_entry_system. The Calculator was a Display and a Keypad, while the

Door_entry_system was a composition of a Keypad, and authentication system, and an actuator. All three of these examples could be implemented with the same method as described for the Robot.

3.2 Requirements Summary

The ten examples that we studied here have many similarities with each other and even some among groups. This is probably because there are very few pure examples of any one of these categories that use only the properties of that single category. Mixing and matching the properties is not a problem because in real-world applications, the systems that use these properties often are not this simple and thus use a variety of these models to complete the solution. Table 3.3 shows the matrix of design examples we have studied to determine the requirements of our inheritance models. The requirements are very similar to the theoretical requirements we discovered in chapter 2 with the addition of some requirements to support interface inheritance.

We can break the requirements listed in table 3.3 into the actual implementation segments. Our implementation consists of POD internals, a bus, the POD manager, and the POD interface hierarchy. We will discuss the implementation details in chapter 4 and walk though our experiments in chapter 5. As shown in table 3.4, each system segment is affected in some way. Most of the requirements are in the POD internals segment, which is where the majority of the work will be done to modify PODs to be able to support the inheritance models we have discussed.

System vs. Requirement	Construction / destruction	Handle acquisition	Exceptions (throw & handle)	POD-POD communication	Virtual methods	Superclass access	Pending messages	Multiple interfaces	Interface inheritance	Callback method	Behavior
ABS_brake	X	X	X	X	X	X	X				
Drivetrain	X	X	X	X	X	X	X				
Smart_microphone	X	X	X	X	X	X	X			X	
Automatic_door_light	X	X	X	X	X	X	X			X	X
Baby_monitor_system	X	X	X	X		X			X		X
Garage_door_opener	X	X	X	X		X				X	X
Touchpad_display	X	X	X	X		X		X	X	X	
Calculator	X	X	X	X		X		X			
Robot	X	X	X	X		X	X		X		X
Door_Entry_system	X	X	X	X		X	X				X
POD extension requirements	X	X	X	X	X	X	X				
POD interface requirements								X	X	X	X

Table 3.3: POD Inheritance Requirements

System segment	Requirements affecting segment
POD internals	construction / destruction, handle acquisition, exceptions, virtual methods, superclass access, pending messages, multiple interface, interface inheritance, callback method
Communication	POD-POD communication
Supporting Classes	construction / destruction, handle acquisition, exceptions, POD-POD communication

Table 3.4: POD System Segments

4. POD SYSTEM ARCHITECTURE

After determining the requirements that our hardware inheritance models impose upon PODs, we came up with an architecture that fits the bill. This architecture is broad and involves a fairly complex interaction between several classes and defines the internal workings of the PODs themselves. Much of the system design was done as a collaborative effort between Frank Sorenson, Darren Hart and the author as shown in figure 1.1, the POD Collaboration Diagram. Because of the nature of this work, the author spent considerable time working with the POD internals design to make sure they were capable of supporting our hardware inheritance models.

Part of the elegance of PODs is their ability to integrate the hardware objects into software programs. We have created several supporting software classes that ease the integration and offer a clean object-oriented API. The classes in the order that we will explain them are: the POD manager, connection classes, proxies, and messages. We will start with an overall view of how the classes are related, then move on to explore each class. At the end of this chapter, after explaining the details of the POD internals, we will discuss the POD system communication mechanism.

4.1 Supporting Classes

An overall view of how the system architecture fits together is shown in figure 4.1. The application, POD manager, connection classes and proxies are all in the same namespace, i.e. that of the host machine. The POD manager contains a pool of connections to PODs. Each POD has exactly one connection. The application creates the POD manager, which then populates its pools. The application instantiates one or more proxies that act in software on behalf of the actual PODs. On creation, a proxy

requests a connection to the appropriate POD. Messages either get created in the proxies or the connections depending on the direction of communication. Connections pass the message contents on to the PODs. The details of the interactions between the classes will become clearer as we further discuss their details.

POD Manager

The POD manager's first responsibility is to keep track of a pool of connections to PODs. Upon startup, it spawns threads for each connection subclass that constantly monitor the sources for new PODs. Each new POD is added to the pool through a process called enumeration, which will be discussed in the POD Internals section of this chapter. Currently, the POD manager is started as a singleton by the application and quits with the application. The application only needs to start the POD manager and has no other need for further communication with it. When the application instantiates a proxy, the proxy requests a connection for the appropriate POD from the POD manager. Connections that belong to the proxies are removed from the free pool to maintain the one-to-one relationship between proxies and connections. PODs can also request a handle to another POD by sending a message to the POD manager with desired type and/or serial number.

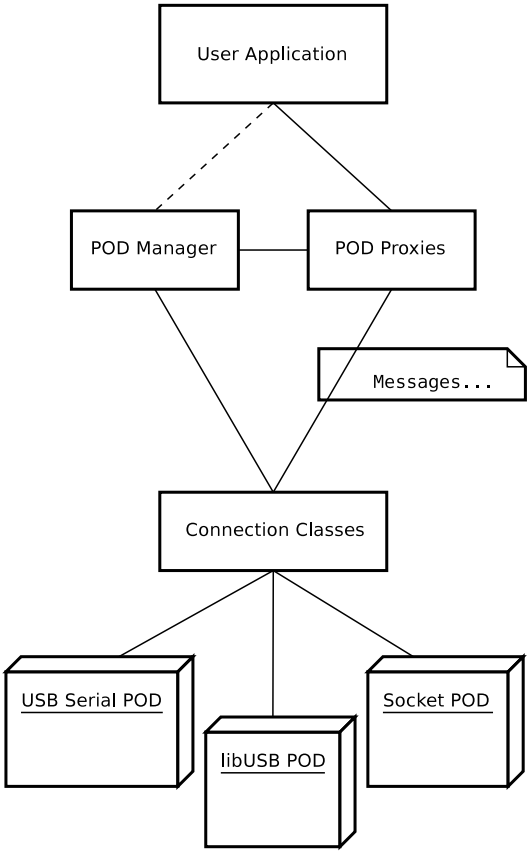

Figure 4.1: System Architecture

This has the same result as a proxy requesting a connection, and the connection is removed from the free pool and added to the ID-connection map.

Facilitating POD-to-POD communication is the POD manager's other main responsibility. It acts as a router for all inter-POD messaging. When a connection receives a message from a POD that addressed to another POD, it hands the message off to the POD manager to forward it to the correct POD. The POD manager looks up the destination connection by ID in the ID-connection map, so the lookup and forward is very fast. Because the USB is a single master bus, only the host can initiate communication, forcing all communication to be routed through the host and thus the POD manager. In a system that used another bus than the USB that has multi-master capabilities, this function of the POD manager could be disposed of, allowing PODs to communicate directly over the bus. A multi-tiered bus might be another advantage, offering the ability to request a POD by subnet rather than serial number. When a POD requests a handle to another POD that is local in its subnet it might not need to use the serial number because it knows that is the only POD of that type in that subnet.

Connection Classes

The connection classes all derive from a connection superclass. Each connection only needs to define the read and write routines for that communications bus. Our implementation of the POD system includes three types of connections: a USB serial connection that uses /dev/ttyUSB[0-15], a libUSB connection that accesses USB devices directly, and a socket connection for TCP/IP PODs and possible "virtual PODs." Each connection class has a static method that is spawned as a thread by the POD manager that monitors its bus and waits for new PODs to be attached. When a new POD is attached, the connection determines the type and serial number

of the POD by issuing an enumerate command as explained in section 4.2, and then has the POD manager add it to the free pool.

Our implementation of connection classes imposes a synchronization measure upon POD communication. Because two of the three currently implemented connection classes use stream-based rather than packet-based communication, the reliability of communication is increased greatly by adding two synchronization bytes at the beginning of each message packet. When the stream from a POD is read, we expect the packet to immediately follow these two bytes. This way, if communication fails, we can synchronize the stream by reading until we have received the two synchronization bytes.

Proxies

Each POD has a software counterpart called a proxy that is essentially a class of stub functions. Each proxy, upon creation, requests a connection from the POD manager for the type of POD it was written for. In addition, proxies can request specific PODs by requesting the connection by serial number. Display proxies ask for connections to a display POD, etc. Similarly, upon destruction, the proxies will release the connection back to the POD manager. For all other member methods, the proxy acts only as a stub for the hardware: it creates a request message object, populates it with the arguments to the method, sends the message using the connection, waits for the reply, unpacks any return values and returns synchronously like a standard member method call. In this implementation, we have pushed all the intelligence as far down to the hardware as possible, so other than the connection itself, proxies should contain no state – they shouldn't cache information or do any processing on their own. For POD-to-POD communication, proxies are not needed or used because proxies are used to interface between the hardware POD and the software application.

Messages

For communication within the software support classes, we have created a message class hierarchy that consists of a base message class with subclasses that correspond to the four message types: request, reply, debug, and exception. These classes are essentially a wrapper around the data packets that the PODs understand as messages. The message classes offer parameter packing routines that manage the packet to simplify the encoding and decoding of messages. The lifetime of a packet is as follows: creation, parameter packing, routing, parameter unpacking, destruction. When it comes time to actually send the message to the hardware, the connection sends the contents of the data packet that is internal to the message class. The PODs have access to similar packing and unpacking routines that are used to decode incoming packets and create outgoing packets.

4.2 POD Internals

Up to this point, we have been discussing the implementation of the software that runs on the host machine. This section focuses on the code that is running on the individual POD microprocessors. Much of the the work done by the author was in direct response to hardware inheritance in the POD internals area. Many of the requirements that we found through our research had to do with how the actual PODs behaved more than the environment they existed in. This section touches all eleven requirements and discusses five of those completely. We wrote a library called libavrpod to assist in development that implements the topics discussed here. We chose the name libavrpod because our target hardware was Atmel microprocessors with the AVR architecture. Much of the library is actually hardware independent and written in the C language.

We start with the basics of how PODs work internally. The libavrpod library defines the `main` symbol as the software entry point, which starts by initializing

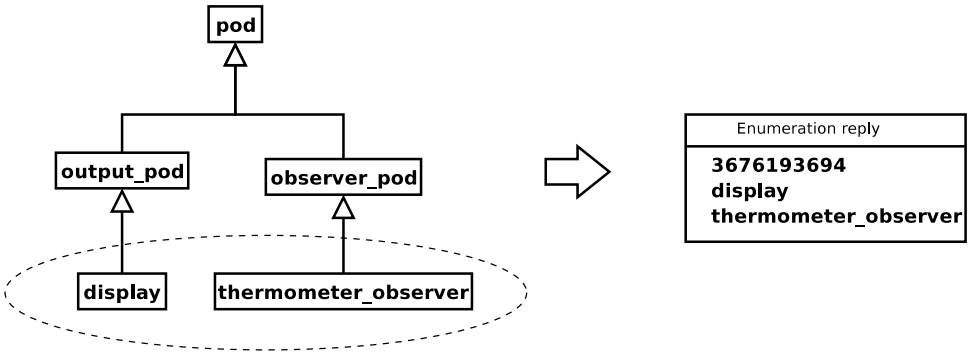

Figure 4.2: Enumeration Example

the communication mechanism and then immediately calls a user-defined method `init` that is used to perform any hardware specific initialization routines. When `init` returns, `main` enters an infinite loop that calls `pod_decode_packet` if a new packet has been received, and then calls `behavior`, which allows PODs to have some sort of behavior. The `init` and `behavior` methods are not as interesting as the `pod_decode_packet` method, which serves as the entry point for all incoming packets, whether they are method calls, exceptions, debug messages, or replies.

The method `pod_decode_packet` behaves differently depending on the POD's current state. To ensure that a single physical POD can only be "instantiated" once, we implemented a simple state machine inside the PODs that behaves differently depending on whether or not it has been instantiated. If a POD has been instantiated, incoming method calls get looked up and executed. Otherwise, if the function call is not a constructor or a call for enumeration, the method is considered illegal and dropped.

Enumeration is the process PODs go through when they are plugged into the bus and get noticed by the bus's connection monitor thread. As previously mentioned, when the connection monitor notices a new device, it sends the enumeration command. This command causes the POD to examine its list of virtual function tables and create a list of devices it can possibly be. This list is composed of all the names

of the leaf members of each interface it implements. As a part of the enumeration reply packet, the POD also responds with its unique serial number. For an example of enumeration, we look at the device shown in figure 4.2, which should simplify this concept. The device can be one of two things, the first of which is a text display. In addition to the text display interface, it fulfills several other interfaces because of interface inheritance. It also implements the thermometer_observer interface, which has its own interface hierarchy. We can determine which interfaces are returned by creating a list of the leaf nodes. In this case, the device is a display or a thermometer_observer. While the device does implement the interfaces that each node requires, only the leaf nodes are concrete classes that can be instantiated. This is why enumeration only returns the leaf nodes.

After the connection manager has enumerated the new device, the connection is added to the POD manager's pool of available connections. When the POD manager receives a request for a connection of a certain type or specific serial number, it looks through this pool at the list of enumerated types and serial numbers for each connection and returns the appropriate one. When a proxy is constructed, it makes this request, receives the connection and then makes the constructor call on the POD. PODs use named constructors since they are often able to implement one of many devices. The name is used internally to search for the correct virtual function table. When a constructor packet is received by the POD, `pod_decode_packet` checks to be sure the POD hasn't already been instantiated and then looks for the appropriate virtual function table. After `pod_decode_packet` has found a table, it executes the construction routine, which may do any number of other hardware or software initialization for the POD. Then, all following commands will be looked up in the virtual function table to determine which function to call.

Often in construction of PODs that use the extension model of hardware inheri-

tance, one POD will need to request the handle of another POD. This is accomplished in a similar fashion to what the proxies do when they request a connection. The POD sends a message to the POD manager requesting a POD of a certain type and/or a specific serial number. The POD manager looks through the pool and returns a handle to that POD. Currently, POD handles are simply a 8-bit unsigned number that is the index to the connection map in the POD manager, while POD serial numbers are a 32-bit unsigned number. Internally, when a POD needs to reference another POD, it sends a message with the destination ID as the handle of the POD. The POD manager can then route the message to the correct POD's connection and to the destination POD.

The POD architecture accounts for four types of messages: request, reply, debug and exception. The messages are all fixed at a length of 64 bytes, with the first four bytes as a header that contains a message ID, a source POD ID, a destination POD ID and a message type. The remaining 60 bytes are the payload of the packet and can be used for parameters, return values, and so forth. The message ID of a request message is set by the sender and any messages in response to that request have the same message ID. The source POD ID is always the POD that the message was created by, regardless of message type. The POD manager and proxies can also create messages and therefore are assigned reserved ID numbers – the POD manager's ID is 0 and the proxies all have an ID of 1. Proxies can all have the same ID because when the connection receives a message with an ID of 1, it forwards it on to the proxy while all other messages get sent to the POD manager. The destination ID is the handle to the POD that the message should be routed to. In the case of a reply, the source and destination IDs are simply swapped. The message type lets the PODs know how to react to each incoming packet and is one of the first things looked at when parsing a packet. Debug messages are more commonly created by

PODs and sent to the POD manager, which prints the contents to its console, but are simply dropped by a POD if it receives them. Exception packets can also go either way. PODs need to be able to respond to some exceptions and can install exception handlers to do so. Throwing exceptions is simply a matter of creating a packet and then sending it to the appropriate destination ID.

As mentioned previously, the method `pod_decode_packet` is the entry point for incoming packets. To determine what to do with a packet, it looks at the message type. Request messages have the first parameter packed as the method ID to call. After looking up the method to call, the packet gets passed and the method unpacks the parameters it requires from the packet. The packing and unpacking library that is in libavrpod and the message class on the host side support a variety of parameter types, including signed and unsigned 8-, 16-, and 32-bit integers, strings, POD handles, method IDs, character strings, and a fixed point 32 bit precision type. The packing and unpacking routines offer some error checking for packet overflow, but the caller must pack the packet correctly or the method will not work correctly.

Each POD has the ability to be several different types, and each type may have polymorphic methods. Part of the reason this architecture is a bit complicated in this area is because hardware is always more difficult to make than software. If there is a piece of hardware that is common to several different interfaces, each interface that can be implemented can be stored in the list of virtual function tables that the POD has. This list of tables is how the POD determines at runtime which constructor to call and what to include in the enumeration reply.

The virtual function tables themselves are a map of method ID to function address that the programmer specifies. Then, at runtime, libavrpod makes a decision to determine which polymorphic method to call. Essentially, it tries to find the function in the subclass it was called on, and then it looks up in turn to each superclass

and executes the first method that has the appropriate ID. In order to search up the chain of superclasses, each subclass has a back-link to its superclass with the ultimate superclass having a `NULL` link. If a method is not found, an exception is thrown to the caller. In software, this is essentially what happens, although this version is simplified a bit and obviously uses a different algorithm. But it does implement the basic concept of polymorphic methods, which is one of the requirements for our system.

A single virtual function table lists all the methods it needs to override over the superclass with a method ID. This ID is a two-byte number with the first byte as the interface and the second byte as the method ID within that interface. The reason method IDs are broken up like this is to allow PODs to implement multiple interfaces. If a superclass implements multiple interfaces, all the subclasses automatically will too by virtue of inheritance and can also override any of these methods by adding the same method ID to the subclass virtual function table. While this was more of a soft requirement for interface inheritance, it is a very convenient thing to have in a developing environment like PODs.

Superclass access was found to be a requirement of most of the examples we have examined. This is done by making a call to the libavrpod superclass lookup method and passing the appropriate parameters. The libavrpod library defines a method signature for all methods that are in the virtual function tables which accepts a packet and the superclass methods would also be of this form, so a "packet" can be forged to be passed to the superclass method that is called. The reasoning behind this is that the binding is still done at runtime and helps reduce code size by not forcing all methods to have wrapper methods that do the packet forging and packing.

The final requirement that was implemented in the POD internals was the ability to have pending messages. Pending messages are what happens when POD A gets a

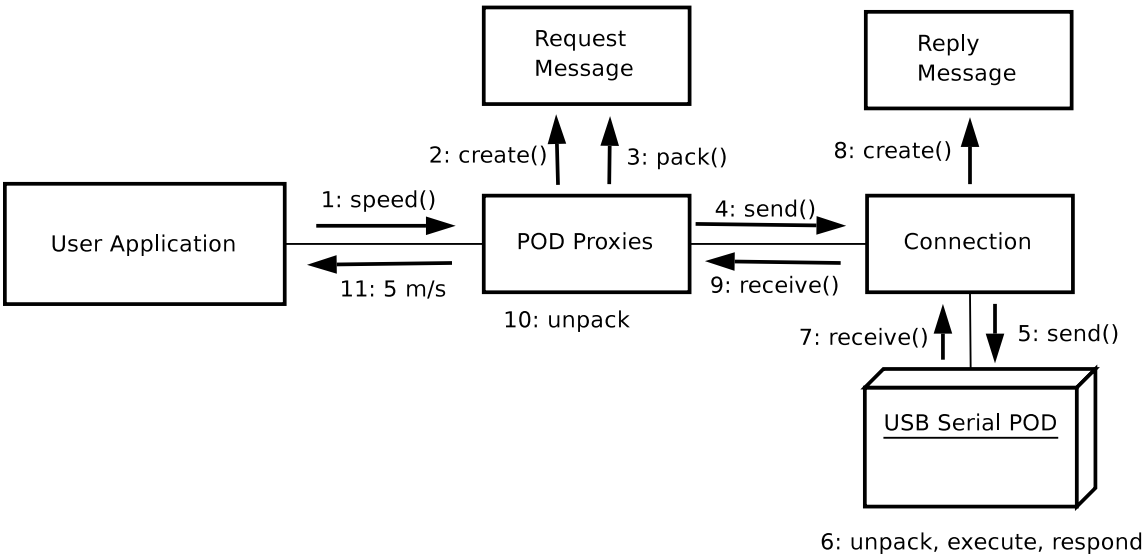

Figure 4.3: Application-to-POD Communication

request to do something that cannot be done without interacting with POD B first. So POD A puts a pending message record in its queue that tells it what to do when it receives the response from POD B. Essentially, this pending message record has a pointer to a callback method that finishes the process that started when POD A received its original request. This sort of interaction is very common in the extension model of hardware inheritance. This mechanism is needed because the POD packet decoding is done on a packet-by-packet basis, which stores no information about previous pending requests that need to be fulfilled.

4.3 Communication

There are two basic forms of communication that the POD architecture supports: application-to-POD and POD-to-POD. Communication involves most of the supporting classes that we have already discussed plus the physical network, which is mostly system independent, but can offer different possibilities based on the network properties. Some networks might offer subnets, multi-master buses, or other features

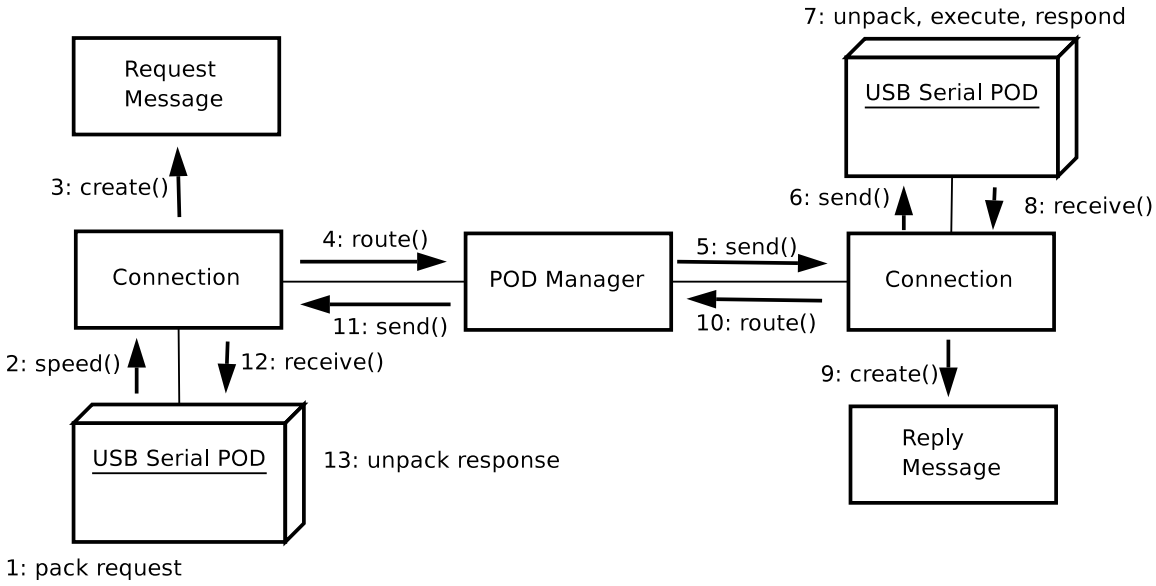

Figure 4.4: POD-to-POD Communication

which could modify the POD manager's role in communication, network bandwidth requirements, and so forth.

In application-to-POD communication, the application starts the process by making a method call on the proxy that corresponds with the POD it needs to communicate with. As shown in figure 4.3, the proxy creates a request message object and packs the method's parameters into the message. The proxy then calls the send method on its connection object, which sends the message's internal packet data over the bus to the POD. The POD receives the message, looks up the method and executes it, unpacking the appropriate parameters from the packet. It then forms a response message and sends it back on the bus to the connection. The connection receives the message and sends it up to its proxy. The proxy, which was waiting for the response, unpacks the message and returns the return values to the calling method in the application. The POD manager is not involved in this kind of transaction at all because proxies have a one-to-one relationship with connections. The POD manager only comes into play with POD-to-POD communication as a routing mechanism.

In POD-to-POD communication, a POD initiates the process. This may be in response to an action, a pending message, or the behavior of the POD. For any such reason, a POD can create a packet and pack the appropriate parameters in it. Then it sends the packet on to its connection. Figure 4.4 illustrates the following example. The connection creates a message object out of the packet and then makes a decision on what to do with it. It sees that the packet's destination ID does not correspond to the proxy, so it forwards the message on to the POD manager for routing. The POD manager looks up the destination in the ID-connection map and sends it on to the correct connection. The connection sends it to the POD over the bus. From here, the exact same thing happens in the other direction. The destination POD parses the message and creates a reply, which gets sent to the connection and forwarded to the POD manager. Then it gets routed back to the first POD through its connection. The POD can then unpack the response and take some action on it. POD-to-POD communication does involve the POD manager, but it does not need to use the proxies because proxies are for interfacing with the software application.

5. EXAMPLE POD INHERITANCE

Chosen Hardware

Implementation of this research meant actually building a few PODs and watching them interact. The hardware we chose to use for our implementations are 8-bit Atmel microprocessors. Our library, libavrpod, was written to work on both the Atmel AT43USB355 and the Atmel ATMega8. The first device is a development board made by Atmel that has a multi-function USB hub with an embedded AVR architecture microprocessor. The second device is a stand-alone 8-bit AVR microprocessor that we built into a circuit that also contained a FTDI USB-serial chip. While the first could offer some very interesting network topologies by using a hub built into a POD, the price and complexity deterred us from completely following that route. Our library ignores the fact that the device is also a hub and uses only one of the other USB functions offered by the chip. This device can be connected to the POD manager using the libUSB connection class. The ATMega8 design uses the USB serial devices /dev/ttyUSB[0-15] in Linux and therefore the USB serial connection class. This design costs about one-quarter of the Atmel development board and offers similar functionality. Most of our PODs were implemented using the ATMega8 design.

5.1 POD Extension

Even simple POD extension tests most of the features that we built into the POD architecture, especially the POD internals. We will start by explaining how our textbook example, the Toggle_feedback_button POD, works so we can see the details and then move on to a more useful and involved experiment, the Thermostat POD.

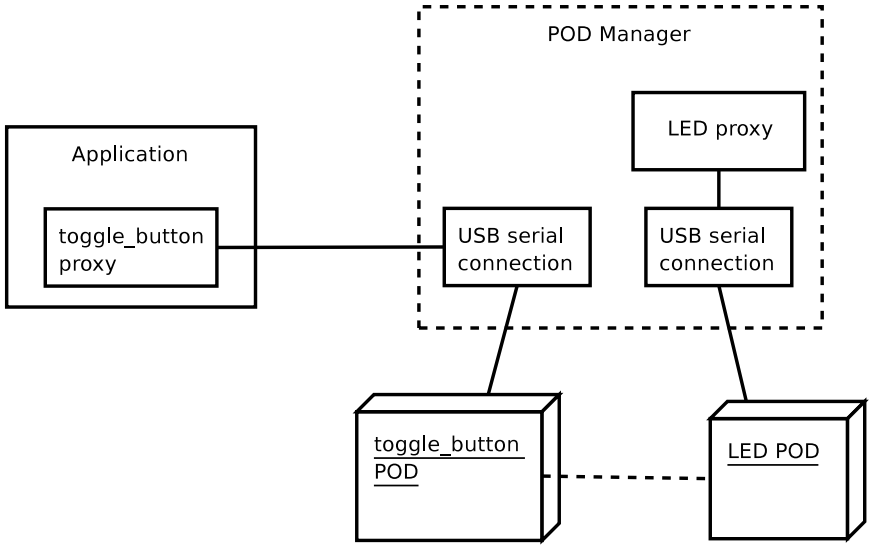

Figure 5.1: Toggle Feedback Button Connections
The dotted line between the two PODs shows the virtual
connection they have through the POD manager.

Toggle Feedback Button

A toggle button is a button that toggles its state each time it is pressed. A very common example of this is a lamp's pull-string switch. When the string is pulled once, the light turns on and when it is pulled again, the light turns off. There are physical buttons available for purchase that do just this. Many appliances have buttons that act like toggle buttons that are really simple contact buttons in conjunction with a microprocessor or a bit or circuitry. Most televisions' power buttons are an example of this – the button toggles the television on and off, but is really often a simple contact button. This is exactly what our Toggle_feedback_button will do. The reason it has feedback in the name is because it will also have an LED that tells the current state of the button. So, the POD should behave as follows: press the button and the LED lights up; press it again and the LED goes out.

We will make this POD by extending a Button POD. A Button POD is a trivial device that interfaces a simple button with a microprocessor so it can be queried

about its state and offer callback methods for when the Button is pressed or released. To add some more state and feedback to the Button, we will add an LED POD, which is simply an output device that can be on or off and can be queried about its state. The interaction between the two PODs is purposely simple to help better explain how this hardware inheritance model works without getting mixed up in the details of the complex interactions of other PODs. Figure 5.1 shows the actual connections and relations between the PODs, the POD manager and the application. The application has a toggle_button proxy that allows it to interact with the toggle_button POD. The proxy has a reference to a connection that the POD manager has established. This connection is the software counterpart of the physical link to the POD.

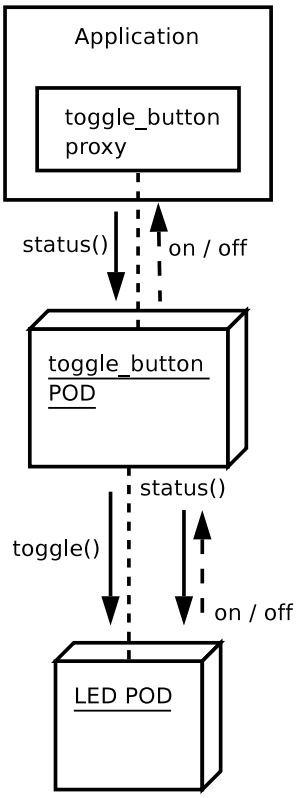

Figure 5.2: Toggle Feedback Button Interactions

When the Toggle_feedback_button is instantiated, it requests the connected LED POD by serial number from the POD manager. Then, it sets up a callback method for `on_pressed`. This method is one that toggles the LED on and off. Figure 5.2 shows the interactions between the two PODs and the application. After the constructor has finished, the application can query the Button to determine its state. This query follows the following procedure: the Toggle_feedback_button decodes the packet as a query about whether or not the Button is pressed. The Button has no state of its own, so it creates a pending message entry to continue execution of this process when it has heard back from the LED POD. It asks the LED POD what its status is and gets a response. This

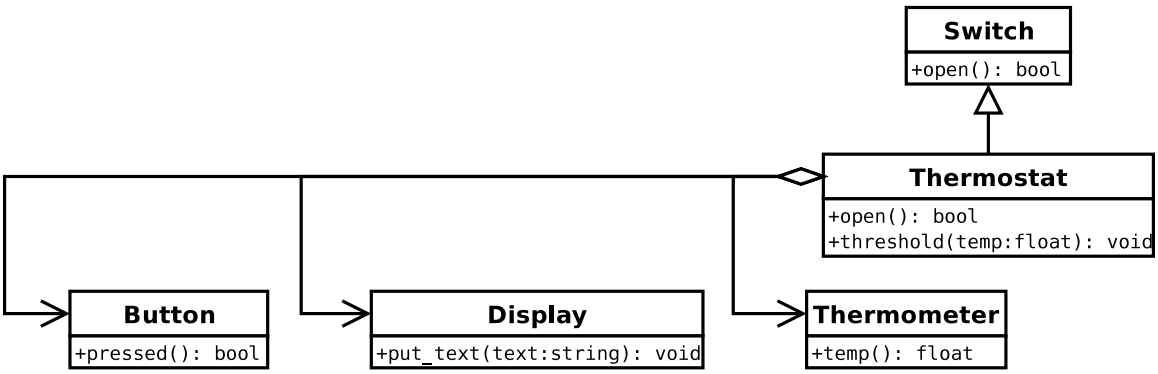

Figure 5.3: Thermostat POD Inheritance

response triggers the pending method to be executed, which then creates a response to the application with the state the LED returned. This process that was executed is really a virtual method that overrides the default Button `get` method. When the Button is actually pressed, the callback method gets executed, which causes the LED to toggle its state so the next time the Button is queried, it will get a different answer. While a thermostat is a bit more complex in components, it follows a similar internal behavior to deal with inheritance and polymorphic methods.

Thermostat POD

A thermostat is effectively a programmable switch that uses a thermometer to determine when to open and close the circuit. If we are dealing with a thermostat for a heater, the thermostat should close the circuit, turning on the heater, when the temperature drops below a certain threshold. The thermostat then opens the circuit when the temperature is above that threshold. A fancy thermostat might even have some sort of interface with buttons and a display and possibly a programmable API to query the state and temperature, set the threshold, etc. This is the kind of thermostat we want to build to show the simplicity involved in doing inheritance in hardware.

We will start by extending a switch POD because we said before that a Thermostat is a programmable switch. To be precise, our Thermostat is a switch that has a

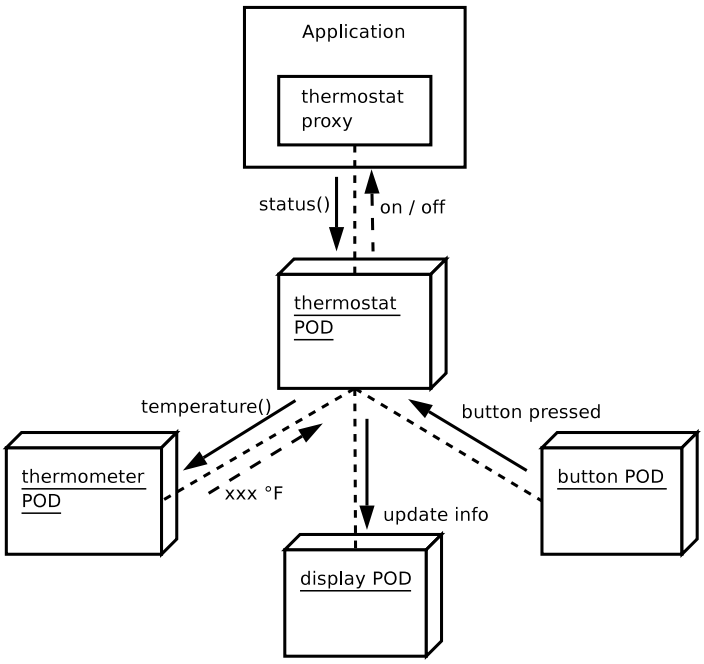

Figure 5.4: Thermostat POD Interactions

thermometer, a display and some buttons. This design gives us an extension diagram (shown in figure 5.3) with the Thermostat extending Switch by adding some Buttons, a Display, and a Thermometer. The Thermostat has a similar connection setup as the Toggle Feedback Button, so we won't show that here. Like the toggle button example, we have a virtual method `open` that returns whether or not the Switch is open. With a Thermostat, we want it to also make sure that the current state of the switch is correct before returning it. To do this, it can query the thermometer for the current temperature and compare that to the threshold to determine the correct switch state, modify the switch state if necessary and then return that.

When the Thermostat is instantiated, it requests the Thermometer by serial number. If the POD manager does not have one, it will throw an exception in the constructor. If found then it will request a Display and Buttons in a similar manner. These, while nice for users, are not required and are used only for show, as all the functionality of the Thermostat is based on the Thermometer. The following inter-

actions are shown in figure 5.4 for help with visualization. The Thermostat defaults to a reasonable threshold, such as 70 ^{o}F. The Thermostat then registers callback methods with the `on_pressed` events for the Buttons. When one Button is pressed, the threshold will go up one degree, while the other Button does the opposite. The Thermostat has a behavior programmed into it that checks the temperature once a minute and adjusts its switch state and Display appropriately. The air temperature in a house is not likely to change too much faster than this rate, although it could be adjusted. When the Thermostat is queried about its current state with the virtual method `open`, it goes through a similar pending message process that the toggle button example above did. It puts a pending message in the queue, and requests the temperature from its member Thermometer. When that method call returns, the pending request is fulfilled and the Thermostat can make a decision as to whether or not it should be an open Switch. It returns this as the answer to the query it received.

The fully functioning Thermostat exhibits all seven of the extension requirements and two of the four interface requirements that we found in chapter 3 – construction & destruction, handle acquisition, exceptions (throw & handle), POD-POD communication, virtual methods, superclass access, pending messages, behavior, and callback methods. This experiment proved to be an effective method of showing the simplicity of hardware inheritance using extension.

Extension Analysis

From research to implementation, we feel that the extension model is a very useful inheritance model when applied to hardware. While examining the design examples, we found that this inheritance model offered us the ability to use proper object-oriented programming methods applied to hardware. We were able to extend devices by defining the differences in a similar manner to software. Code and hardware reuse are two other benefits that this model offers.

While we made the process of extending a POD as simple as possible, it still means that the code running on the embedded microprocessor must be modified, which means this is not a task for the novice. Using PODs that have already been programmed for extension, however, would be a trivial matter. We will discuss the code that is required and the connections to be made with an example extension in appendix A.

Other possible benefits of extension are gained by changing the bus. Using a multi-tiered bus, we can gain a lot in terms of lowered bandwidth use and a simpler POD manager implementation. If each POD to be extended had an upstream port and a downstream port, all communication that goes on within the extended POD would be done on the subnet, thus reducing the traffic on other parts of the network. The POD manager could be simplified and run in a distributed manner on each device that has a downstream connection.

5.2 Interface Inheritance

Interface inheritance is a fairly straightforward thing to test. Any given POD has an inheritance path back to the base POD interface. Along each class in the path, the interface for that class should be implemented and can be tested. Interface testing is an important part of ensuring that PODs can be interchangeable. We have defined a hierarchy of devices and classes, that can be found in appendix B. This set of classes can be extended as people find more types of PODs to implement. We will use this tree to do our interface testing.

Interface Testing

Every POD exports a basic POD interface that can be used to determine a URL where more information about that device can be found. This URL should, among other things, show the interface hierarchy for that POD. Then, using that information, we can test each interface to ensure they are all completely implemented. We wrote a

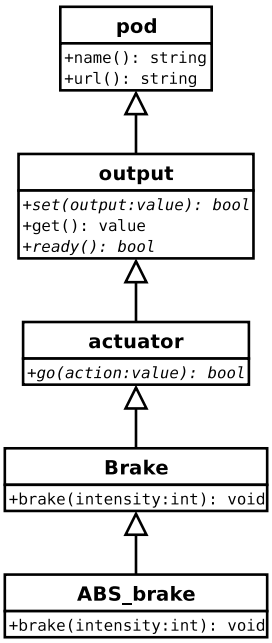

Figure 5.5: ABS_brake Interface Hierarchy

---

program in C++ that parses two XML files: the POD interface file that describes the specialized interface for the POD, and the POD interface hierarchy file that contains the API for each interface in our interface hierarchy.

We will walk through a testing example to show how this works. The ABS_brake's complete inherited API is shown in figure 5.5. The specialized interface file for the ABS_brake needs to specify the ABS_brake interface and specify that it implements the Brake interface. From this point on, all the interface information is found in the POD interface hierarchy. The program can then walk through each interface up to POD and test each method in those interfaces. If the POD throws a `method_not_found` exception, we know that the interface is not fully implemented. Naturally, this also depends on the POD to throw an exception when a method is called on it that is not part of its interface. Our library, libavrpod, implements this as part of the mechanism that does the method call lookup.

Interface Inheritance Analysis

We have found in our research and in our implementation that the interface inheritance model is very useful. This seems to be a standard kind of thing to do; many programming environments offer interfaces to implement and inherit from. Making interchangeable drop-in replacements for PODs makes it easier for users to find and use PODs. We also found that the interface inheritance model works well alongside extension as well as a stand-alone model. When a device has already implemented all the required interfaces, inheriting from that device ensures that the new device also implements all the required interfaces.

6. CONCLUSIONS AND FUTURE WORK

6.1 Requirements Summary

Until now, object-oriented hardware lacked some of the features, such as inheritance, that make software objects so useful and powerful. Finding parallels in software and hardware development helped us to implement two prevalent software inheritance models in hardware objects. After researching inheritance models in software and examining the research of others in the area of hardware objects, we decided that the POD architecture was the closest to the goal of hardware inheritance. We looked at ten design examples to help determine the things we would have to add to the POD architecture to allow it to support inheritance. We found that in order to make inheritance in PODs work, we needed to implement the following requirements:

- Construction and destruction
- Handle acquisition
- Exceptions (handling and throwing)
- POD-to-POD communication
- Virtual methods
- Access to superclass methods
- Pending messages
- Multiple interfaces
- Interface inheritance
- Callback methods
- Behavior

After implementing these additions to the POD architecture, we created some examples to show that the implementation worked. The extension and interface

inheritance models proved to be just as useful in setting up hardware systems as when applied to creating a complex software system. We found that these eleven requirements were sufficient to support our examples and thus should be able to create the exploratory design systems we used to determine the requirements. These exploratory designs were chosen as a possible subset of the types of systems that PODs could be used for and show us how useful these two inheritance models can be in hardware. The exploratory designs showed the advantages of using inheritance models with PODs over the previous structural programming models. The simplicity of the designs, the time saved, and the parallels with software that these designs exhibited were sufficient proof to validate our architecture. The experiments that we did also show simplicity in design, abstraction, code and hardware reuse and other benefits of inheritance. We feel that our work has shown the benefits of software inheritance can also be found by applying these inheritance models to hardware.

6.2 Future Work

The POD architecture, still in the defining stages, offers many directions for future research and development. As this research showed, bigger and better PODs are already on the way. Some of the topics future research could include are: applying software design patterns to PODs; exploring the composite model with PODs; designing a PODs simulation environment; adding support for new buses, topologies, and protocols; adding support for real-time handling of POD events; and miniaturization. These are just a few of the areas of work that the extended POD architecture can offer.

In addition to work on the POD architecture itself, there is much that can be done with PODs themselves. Rapid development and prototyping is one area that PODs could excel in. Because of the modular nature of PODs, they can be switched in and out of test beds, extended to create more complex PODs, and easily programmed.

A. POD DEVELOPER'S GUIDE

This appendix explains in detail the steps to extend a Button POD to make a Feedback_toggle_button POD. There are two parts that need to be written – the libpods (host) portion, and the libavrpod (device specific) portion. With extension, there is no change in how the PODs are connected because we are using the USB as our bus; all PODs are simply connected straight to the host machine rather than to each other. The POD manager provides a virtual connection from POD to POD.

This example is intended to show the simplicity of extending a POD. Our architecture was designed to minimize the steps to take for extension. We have pushed the intelligence of the PODs as far down toward the hardware as possible, so the PODs do the virtual function lookups at runtime. This means that to extend a POD we must modify some code that runs on the microprocessor. We have simplified this process by creating several examples of PODs and extensions. Figure A.1 shows an overview of the things that need to be changed in the extension process.

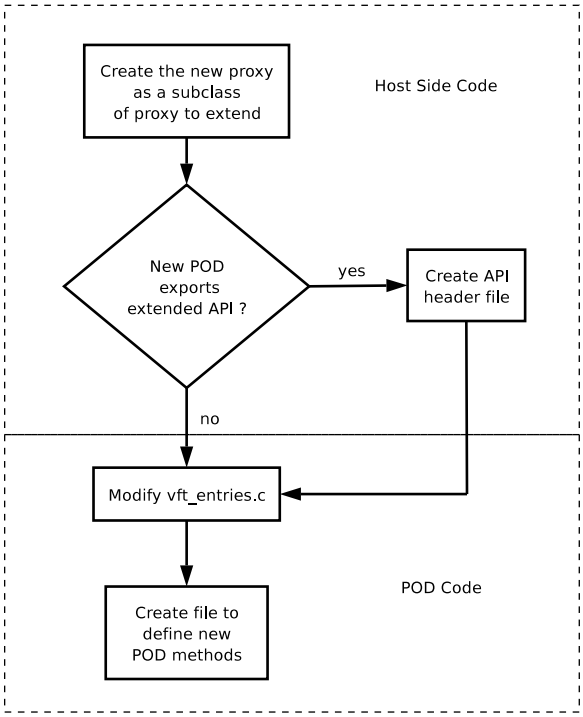

Figure A.1: POD Extension Flowchart

```
1  class toggle_button_proxy : public button_proxy
2  {
3      public:
4          typedef boost::shared_ptr<toggle_button_proxy> ptr;
5          typedef boost::weak_ptr<toggle_button_proxy> weak_ptr;
6
7      protected:
8          toggle_button_proxy() { }
9
10     public:
11         static toggle_button_proxy::ptr create(uint32_t serial=0);
12         virtual ~toggle_button_proxy() { }
13         virtual std::string name() const { return "toggle_button"; }
14
15         // virtual uint8_t get();
16
17 }; // class toggle_button_proxy
18
19 toggle_button_proxy::ptr toggle_button_proxy::create(uint32_t serial)
20 {
21     // construct the proxy and get a connection object
22     toggle_button_proxy::ptr tb_proxy(new toggle_button_proxy());
23     // get the conn_ from pod_manager
24     tb_proxy->conn_ = pod_manager::get()->request_pod("toggle_button", serial);
25     if (!tb_proxy->conn_) throw no_conn_exception();
26     // inform the connection object of the proxy
27     tb_proxy->conn_->set_proxy(tb_proxy);
28
29     // send the constructor message to the POD (returns void)
30     request_message::ptr constructor =
31         request_message::create(tb_proxy->next_msg_id(), POD_CONSTRUCTOR);
32     constructor->pack_string("toggle_button");
33     tb_proxy->conn_->send_message(constructor);
34     reply_message::ptr reply = tb_proxy->wait_for_response(constructor->id());
35
36     return tb_proxy;
37 }
```

Figure A.2: Toggle Button Proxy

A.1 Host Side Code

Libpods, the host machine library that defines the POD manager and the other supporting classes, is where the proxy classes reside. A new proxy must be created that exports the Feedback_toggle_button API. The code for this proxy is simple because much of the work has already been done in the superclass Button proxy class. As figure A.2 shows, the only code necessary is for construction of the POD. In our architecture we used a named constructor called `create` to do the construction. Part of the reason for this is because we didn't want the superclass to grab the wrong

```
1  #ifndef BUTTON_API_H
2  #define BUTTON_API_H
3
4  #include "input_api.h"
5
6  #define BUTTON_API 0x05
7
8  #define BUTTON_IS_PRESSED   COMMAND_(BUTTON_API, 0x01)
9  #define BUTTON_ON_PRESS     COMMAND_(BUTTON_API, 0x02)
10 #define BUTTON_ON_RELEASE   COMMAND_(BUTTON_API, 0x03)
11
12 #endif
```

Figure A.3: Button API Header

connection or send a constructor command to the POD before the derived class's constructor was called. Note that on line 15 of figure A.2 the `get` method is commented out. We point this out to emphasize that because the virtual function lookup is done on the POD, there is no need to override the `get` method on the host side. Here in the proxy would be where any other methods that extended the interface of the Button would go if desired.

When the extended POD exports a different interface than the superclass, an API header file must be created to specify the interface ID and the method IDs for that interface. The button_api.h file is shown in figure A.3. This file includes the API header file for the input class, which in turn includes the API header file for the pod class. This allows us to have access to all the classes and their API ID numbers up the interface hierarchy. Each command is generated with the `COMMAND_()` macro which generates the 16-bit method ID from the 8-bit interface ID and command ID numbers. These numbers are how the proxy and POD determine which method to call. Creating a proxy and possibly creating an API header file is all that is needed on the host for an extension with a new API.

```
1  /******** From vft_entries.c ********/
2
3  // include externs for all user defined pods
4  extern const vft_t button_vft;
5  extern const vft_t toggle_button_vft;
6
7  const vft_t * vft_list[] =
8  {
9      &button_vft,
10     &toggle_button_vft,
11     NULL
12 };
```

Figure A.4: VFT Definitions

A.2 POD Internal Code

On the device side, we need to add code to the POD that we are extending. Our library, libavrpod, is written in C and has been written in a way that makes extending a POD as painless as possible. Each POD has at least two files that define its behavior. Figure A.5 shows parts of both of these files. The first file for each POD is called vft_entries.c. This file contains an array of `vft_t` structs that lists all the `vft_t` structs for the POD. Each constructible interface has a `vft_t` struct that contains information about that device as well as the vft_entry table. The reason this file is separate from the main code is because the variable `vft_list` defined on line 7 of figure A.4 is required by libavrpod and there should only be one that contains the information for all possible PODs for that device (in this case it lists button_vft and toggle_button_vft.) The other reason this is a stand-alone file is to make it possible to extend a POD without touching the actual superclass code. In this way, object files could be distributed with this single vft_entries.c file and Joe User could extend the POD without access to the proprietary source code for the superclass.

The second is a file named after the POD, in our case this is toggle_button.c, that implements each method in a way that can be called from the `pod_decode_packet` method. The signature for this kind of method is defined by `typedef packet_t*`

```
1  /******* From toggle_button.c *******/
2
3  static const vft_entry_t toggle_button_vft_entries[] =
4  {
5      /* first entry must be constructor */
6      { POD_CONSTRUCTOR, toggle_button },
7      { POD_DESTRUCTOR, toggle_button_done },
8      /* button interface */
9      { INPUT_GET, get },
10     /* last entry must be NULL */
11     { 0, NULL }
12 };
13
14 static const char toggle_button_constructor_name[] = "toggle_button";
15 static const char toggle_button_url[] =
16     "http://pel.cs.byu.edu/~sorenson/PODS/button";
17
18 // declare button's vft so we can "extend" it
19 extern const vft_t button_vft;
20 const vft_t toggle_button_vft = {
21     toggle_button_constructor_name,  // pod name
22     toggle_button_url,               // pod url
23     &button_vft,                     // superclass POD
24     toggle_button_vft_entries,       // vft_entry table
25     behavior                         // callback for behavior
26 };
```

Figure A.5: VFT Definitions

`(*pod_function_t)(packer_t *)`. This tells us that they take a `packer_t *` and return a `packet_t *`. The `packer_t` is the struct through which packing and unpacking of parameters is used. The `packet_t` is a struct that defines the various parts of the packet, e.g. message id, source, destination, type, and body. Each method unpacks the parameters that are required, performs some action, and creates an appropriate response. All of these methods are defined as static for protection and to avoid namespace pollution. To access the POD's methods, libavrpod reads a table that is created with method IDs and function pointers. We call this structure the vft_entry table. Line 9 of figure A.5 shows the entry for the `get` method, which overrides the Button POD's `get` method. The comments in the figure explain the entries for the `vft_t` struct. In our case, we want to create another file called toggle_button.c that defines our constructor, destructor, and all the methods that we are overriding. In our case, we need to define `toggle_button`, `toggle_button_done`, and `get`.

A.3 POD Extension Summary

Extending a POD can be done in two straightforward parts. First, we create a proxy class that extends the proxy for the POD we wish to extend. The only thing this class really needs to do is request a connection from the POD manager and send the constructor command to the POD. Any other API extensions also need to be defined here. If there are any API extensions, an API header file must be created with the interface ID and the method IDs for the new methods. Second, we get into the POD runtime code and create a new file to hold our new function definitions. This file contains our constructor, destructor, any other methods we override, and any new methods our API exports. After creating the new file, we modify vft_entries.c and add our `vft_t` struct to the array. After making these changes, we compile and program the device. After all the modifications have been made, the extended POD can be accessed through its proxy just like any other POD.

B. POD INTERFACE HIERARCHY

The POD interface hierarchy was created to define a standard set of POD categories and sub-categories that PODs can fit into. We started by looking at some common devices that might be used as PODs. From there, we found that most of the devices could fit into `input` and `output` classes. These two classes form the second level in the hierarchy. Next we set out to find some common uses for the devices we looked at to determine what API each device should have. This included looking at other non-POD hardware objects and their interfaces.

The resulting hierarchy is shown below in figure B.1 with more detailed views of the input and output branches of the tree shown in figures B.2 and B.3 respectively. Some of the devices shown in the figures, such as `led` and `thermometer`, do not extend their superclass API because the interface is already sufficient for the device. They are shown in the tree to suggest where such devices belong.

By following the hierarchy from any given interface up to `pod`, it is possible to determine exactly which interfaces must be implemented. For example, a POD that implements the `graphical` interface must also implement the `display`, `output`, and `pod` interfaces as well in order to be superclass or upward compatible.

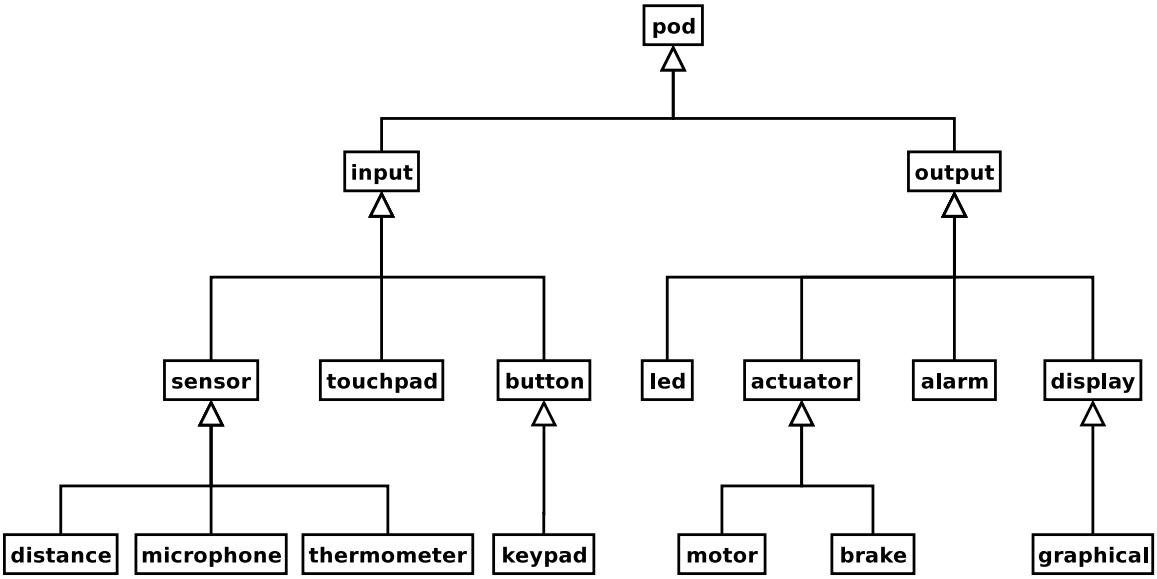

Figure B.1: POD Interface Hierarchy

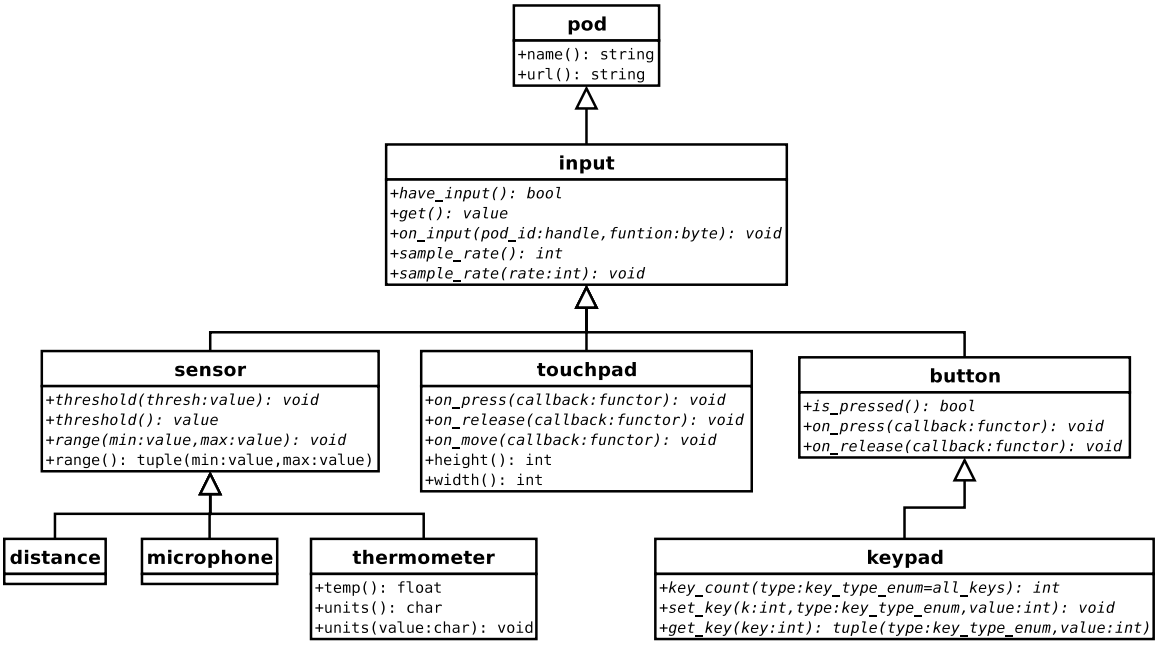

Figure B.2: Input POD Interface Hierarchy

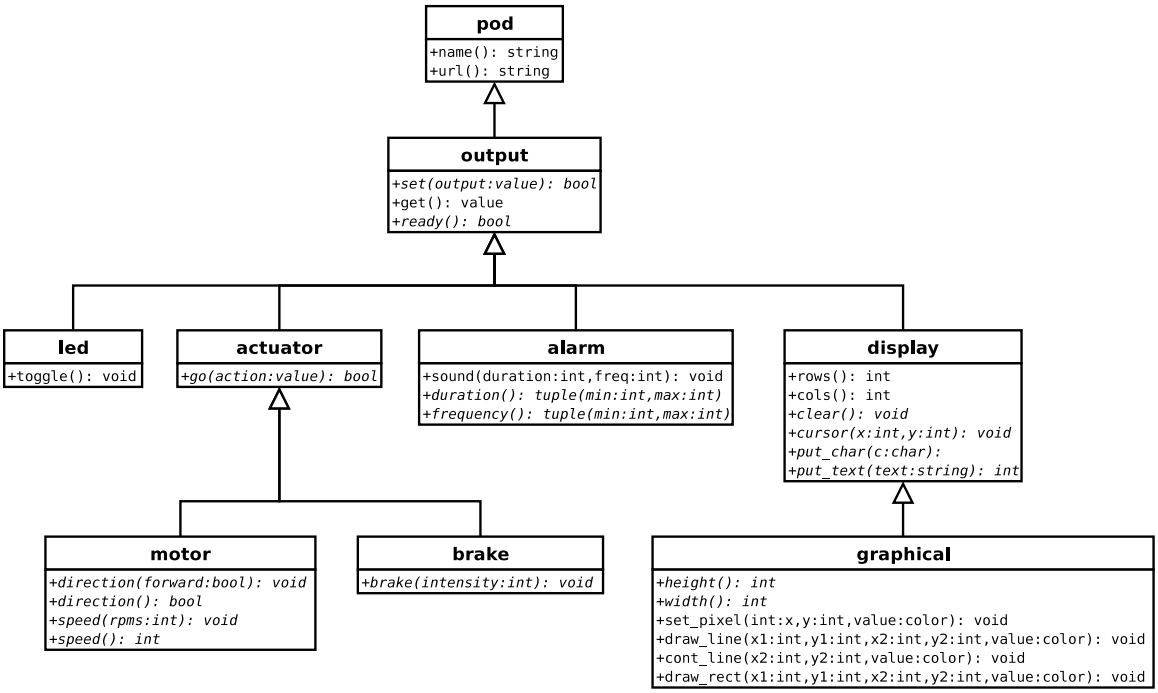

Figure B.3: Output POD Interface Hierarchy

LIST OF REFERENCES

[1] P. Craig Hollabaugh, *Embedded Linux*, 1st ed. Indianapolis, IN: Addison-Wesley Pearson Education, 2002.

[2] M. Kyng, "Centre for pervasive computing," 2003. [Online]. Available: http://www.pervasive.dk/

[3] F. Sorenson, "Object-oriented hardware: Physical object devices," Master's thesis, Brigham Young University, 2004.

[4] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software.* Prentice-Hall, Inc., 1990.

[5] T. L. Group, *Lego Mindstorms, Robotic Invention System, System 2.0.* The Lego Group, 2002.

[6] "Oopic home page." [Online]. Available: http://www.oopic.com/

[7] S. Greenberg and C. Fitchett, "Phidgets: Easy development of physical user interfaces through physical widgets," in *Proceedings of the ACM UIST 2001 Symposium on User Interface Software and Technology*, Orlando, Florida, Nov. 2001.

[8] D. V. Hart, "Using hardware objects in object oriented software design," Master's thesis, Brigham Young University, 2004.

[9] J. Durham, "History-making components," Apr. 2001. [Online]. Available: http://www-106.ibm.com/developerworks/webservices/library/co-tmline/

[10] J. Martin, *Principles of Object-Oriented Analysis and Design.* Prentice Hall, Inc., 1993.

[11] B. J. Cox, *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley Publishing Company, Inc., 1994.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*, 1st ed. Indianapolis, IN: Addison-Wesley, 1995.

[13] A. L. Winblad, S. D. Edwards, and D. R. King, *Object-Oriented Software.* Addison-Wesley Publishing Company, Inc., 1990.

[14] Atmel, *Full-speed USB Microcontroller with Embedded Hub, ADC and PWM: AT43USB355. Revision 2603B USB*, 2002.

[15] Compaq, Intel, Microsoft, and NEC, *Universal Serial Bus Specification. Revision 1.1*, Sept. 1998. [Online]. Available: http://www.usb.org/developers/docs/

[16] D. Fliegl, "Programming guide for linux usb device drivers," 2003. [Online]. Available: http://usb.cs.tum.edu/usbdoc/

[17] S. J. Gowdy, "Linux usb," 2003. [Online]. Available: http://www.linux-usb.org

[18] unknown, "I2c, i2s. – the educational encyclopedia," 2003. [Online]. Available: http://users.pandora.be/educypedia/

[19] G. Kroah-Hartman, "How to write a linux usb device driver," Oct. 2001. [Online]. Available: http://www.linuxjournal.com/article.php?sid=4786

[20] R. M. Neswold Jr., "A gnu development environment for the avr microcontroller," 2003. [Online]. Available: http://users.rcn.com/rneswold/avr/

[21] G.-M. Hoydalsvik, E.-A. Karlsson, S. Srumgrd, S. Thunem, and E. Tryggeseth, "Object-oriented development with and for reuse," Oct. 1991. [Online]. Available: http://citeseer.nj.nec.com/hoydalsvik91objectoriented.html

[22] G. F. Templeton, "Object-oriented programming of integrated circuits," *Communications of the ACM*, vol. 46, no. 3, pp. 105–108, Mar. 2003.