

OOPSLA 2001 Workshop: Towards Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems

A Proposal for an Embedded Systems Pattern Language

David E. DeLano

Object Technology International, Inc.

David_DeLano@oti.com

Background

I consider myself an embedded systems developer. The first 20+ years of my career were spent working on telephony systems, definitely qualifying as distributed embedded real-time systems. I have been active in the Patterns community for around six years. After two rounds at AG Communication Systems that totaled fifteen years, and the rest with a small telephony startup, I made a career change earlier this year. I am now employed with OTI, working on Java Virtual Machines for embedded systems. Part of that work consists of supporting extensions that implement the Java real-time specification.

Introduction

Most of the current Patterns and Pattern Language work on real-time embedded systems has centered on resolving specific problems, or specifying architectures. A Pattern Language for Distributed Real-time and Embedded Systems should be more encompassing and more general. It should start from ground zero, and build a system up from scratch. This is not to say that the existing Patterns and Pattern Languages are bad. They usually start somewhere in the middle, and the user must know a great deal about real-time embedded systems before these Patterns and Pattern Languages make sense and become useable.

On the other hand, I'm confronted with The Real-Time Specification for Java, which essentially is a specification for an interface for a real-time system. This specification has been implemented in Virtual Machines destined for embedded systems, whether or not the specification intended this. The design of the specification could have benefited greatly from the existence of a Pattern Language for real-time embedded systems.

Then there is the debate of whether or not to use the Object Oriented paradigm when developing embedded system. The last few projects I worked on used C++, though I must admit that most designers only used it as a better C. A project before that used a homegrown object oriented C, before cross-compilers for C++ were available. My current work is in Java. I think the Object Oriented paradigm is here to stay, but one must always remember to use all the tools available when solving a problem, and not use OO where it is not appropriate.

So what has Object Oriented embedded real-time development given us? When done correctly, it has given us reusable architectures, reusable designs, frameworks, and better understood code. When misused it often gives us bigger messes than we had before. Object code can be more difficult to test and, especially, debug. Inheritance hierarchies can be so deep that it is difficult to figure anything out. The good, however, most likely outweighs the bad. Object Oriented design and implementation have been especially helpful in the handling of data. Data in Structured Programming usually lies naked to the world with only agreements between developers to protect it.

The Beginnings of a Pattern Language

This proposal for the beginnings of a Pattern Language is by no means complete. It only starts things in the direction that I think they should go. It starts as a Pattern Language for Embedded Systems. It extends into distributed systems and real-time systems. It is not inherently Object Oriented, at least at first, but Object Oriented architectures would come into play as the language matures. To tie this language into the work I am currently involved in, I will pull in the relationship of the Java Real-Time Specification to the Pattern Language, including the proposal of one Pattern for the Pattern Language.

Hardware/Software

Pattern Languages are about defining and dividing space. In most Pattern Languages, it is the definition of "space" that causes the most problems. I view "space" as a broad, generic term. It can even be time based, or action based. In this problem space, though, things look to be fairly self-explanatory, at least in the initial phases. The most obvious separation of space is Hardware and Software. We can divide that part of the problem space which we can see and touch as Hardware, and the part that we know is there but can't prove it as Software. It is a division between that which is more or less permanent, and that which is usually changeable.

Whenever something is divided, tension occurs along the borders. It is in the resolution of this tension, or forces, that Patterns appear and the Pattern Language is pushed forward. Obvious tension occurs in the decisions as to what goes into Hardware and what belongs in Software. Forces also exist in the individual Patterns themselves, causing more sub-divisions, which further fill in the Pattern Language. For example, we could have a Pattern, Firmware, that resolves the issues of having software that we really want to make a more permanent part of the problem space. We can still probably change it, but it is very tightly coupled to the Hardware.

As divisions of the problem space are made and the forces are resolved, places will be made for existing Patterns and Pattern Languages in the embedded systems domain. This includes many of the Patterns in *Pattern-Oriented Software Architecture: A System of Patterns* [1], *Design Patterns* [2], and the various volumes of *Pattern Languages of Program Design* [3,4,5,6].

Hardware has such Patterns as Data Storage and Processor. Data Storage is further refined as ROM, RAM, and Disk, for starters. There are Patterns associated with each of these that help determine which to use, and how large they need to be. Processor is a little more nebulous as it is really a given. There may be patterns on how to choose a Processor.

The most obvious division of the Software domain is the Operating System and the Application. The Operating System covers a wide spectrum of things that could include Drivers and Protocol Stacks. The purpose of the Operating System is to hide as much of the Hardware domain from the Application as possible, only exposing what is necessary. The Application is the top layer that is the essence that gives the embedded system its defining characteristics. Most embedded systems run a single Application, though we are seeing a stretch of embedded systems toward desktop capabilities. For all practical purposes, a handheld PDA is an embedded system, yet it can run various applications the way a desktop would.

Between the Operating System and the Application is another border. This border creates many forces that have to be resolved in order to have a working system. Many systems have been developed with the Application communicating directly with the Operating System. This arrangement has its drawbacks, though, and the two are better off separated by an OS Abstraction Layer.

OS Abstraction Layer

Writing an Application to run on a specific Operating System makes a tightly coupled system that is difficult to reuse. The Application, or even parts of it, does not easily port to any other Operating System. The Application may also be bound to the Hardware if the Operating System does not fully hide the Hardware interface. It may be advantageous to bind the Application to the Operating System, and possible to the Hardware to achieve a real-time efficient system. It can also reduce the memory requirements of the system. More often than not, though, there is enough processing power and memory in current embedded systems to support more abstraction, and thus provide an Application that can be used in future systems.

Therefore, use an OS Abstraction Layer between the Application and the Operating System.

The OS Abstraction Layer should provide an interface for the major responsibilities expected of an Operating System. It can supplement the Operating System for features that are missing, but in general, it should not add any functionality that is not generally expected. For the most part, the OS Abstraction Layer standardizes parameter data, congregates Operating System atomic operations into useful functions, and makes the Operating System behave in a predictable manner.

Now, all that is needed to move to another Operating System is to port the OS Abstraction Layer. Better yet, there are off-the-shelf OS Abstraction Layers, often existing as Frameworks, that are available. Many are available on multiple Operating Systems, so the porting work is done for you. Most of these present a simpler, more uniform representation of the Operating System interface than relying on direct calls.

There can be drawbacks with this solution. Using an OS Abstraction Layer will inherently effect the real time and memory usage of the system. However, Processors are getting faster, and memory cheaper. For the most part, it might boil down to Hardware design.

Most off-the-shelf OS Abstraction Layers have an associated cost. Some are tied in with a framework, and are not easily accessed if the entire framework is not appropriate for your application. Writing your own OS Abstraction Layer is an option. However, this can also become costly unless you are really an expert at it. Some home grown OS Abstraction Layers are developed because of political issues. The designers don't trust the OS or think it needs additional features. Lead designers don't trust other designers to use the OS properly. These OS Abstraction Layers usually lead to something that is not standard and not portable, both which are key elements to the use of an OS Abstraction Layer.

OS Abstraction Layer is similar to MicroKernel [2]. It can also be looked at as a Façade [2] or Adapter [2] for the Operating System.

POSIX compliant Operating Systems are an example of implementing an OS Abstraction Layer through the use of a standardized interface. ACE contains an OS wrapper that is a good example of an OS Abstraction Layer. Though Java is a language, it hides many of the interfaces to the Operating System. This is further enhanced by The Real-Time Specification for Java.

The Real-Time Specification for Java breaks the OS Abstraction Layer into the following domains, which could be considered for Patterns: Threads, Scheduling, Memory Management, Synchronization, Time, Timers, Asynchrony, Systems and Options, and Exceptions. These may not be the most ideal categories for Patterns, and they may be incorrectly grouped, but they do form a proposal from which to expand. The development of the specification would have benefited from a Pattern Language, if one existed. Instead, it now serves as a known use.

To progress in the development of Patterns that surround OS Abstraction Layer, the Java real-time specification should be compared and contrasted to other examples of OS Abstraction Layers such as POSIX and the ACE OS wrappers. This comparison should go a long way in determining what needs to be in the Pattern Language, and what can be omitted. It can also find Patterns that are missing in one or more of the examples. Identifying these short-comings can then help to refine the examples.

An example of a problem in the Java specification is that Scheduling can be done in such a way that if the requirements needed by a task cannot be met, the task is not allowed to run. This might be fine in some systems, but in embedded systems, this should instead result in some sort of recovery. This could be in the form of task or resource throttling. A complete refusal to run is usually unacceptable. Patterns or a Pattern Language on Scheduling would have helped guide the development of the specification.

Conclusion

I would like to see a Pattern Language developed that covers the essence of building embedded systems. This Pattern Language should take into account existing literature, as many of the Patterns already written should find homes within the Pattern Language. I am especially interested in work on the part of the Pattern Language that would flesh out OS Abstraction Layer. This could aid in implementation of the Java real-time specification, or in the refinement of the specification. As the borders of the Pattern Language are refined, it would need to pull in Patterns related to real-time and distributed systems. However, well designed embedded systems are often easily extended to be real-time efficient and behave well as elements of a distributed system. Using a Pattern Language to guide in the design would help ensure this success.

References

- [1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley and Sons, 1996.
- [2] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [3] J. Coplien, D. Schmidt (eds.), *Pattern Languages of Program Design*, vol. 1, Addison-Wesley, 1995.
- [4] J. Vlissides, J. Coplien, N. Kerth (eds.), *Pattern Languages of Program Design*, vol. 2, Addison-Wesley, 1996.
- [5] R. Martin, D. Riehle, F. Buschmann (eds.), *Pattern Languages of Program Design*, vol. 3, Addison-Wesley, 1998.
- [6] N. Harrison, B. Foote, H. Rohnert (eds.), *Pattern Languages of Program Design*, vol. 4, Addison-Wesley, 2000.