Object Oriented Design for Embedded Software Engineers

James Grenning

San Jose, April 2007, ESC Class# 209

The world is made of objects. People think in objects. Objects will make programming simple. Objects will save the world. These are all silly statements that exaggerate the hype around objects. There is hype for sure, but there is also legend. Objects are too slow and fat for embedded software. OO tools are not available for embedded programming.

Objects are not a panacea, but Objects and the concepts of Object Oriented Design are useful in designing embedded software. In this paper I will try to go beyond the hype and explore a few engineering practices and principles of Object Oriented Design, and how to take advantage of them in embedded software development.

There are two "old as the hills" concepts in software development, coupling and cohesion. Coupling between modules should be loose, and cohesion within a module should be high. Loose coupling and high cohesion can be accomplished in the most popular embedded systems programming language, C, but the language gives the programmer very little help in building modular systems. We might believe that loose coupling and high cohesion are good, but why? What problems are being solved? There is a short answer, change!

Change

Software is supposed to be soft; it is supposed to be easy to change. It is only made up of files in a computer and files are easy to changes so software should be easy to change, right? Wrong, software is becoming more and more complex; our embedded devices are becoming more and more complex. My cell phone in the early 90's was just that, a cell phone with a few speed dial numbers. Now it has all my contacts, names, address, phone numbers and email addresses. Being a fully functional PDA was not enough, so now the cell phone has a camera, a music player and video player. As new features are added, the old features, like making phone calls, are still expected to work. Change is inevitable. I bet your company would rather not re-invent the features in the last product just because some new features are being added.

Features are not the only thing changing. Embedded software engineers have to expect hardware changes as well. Such as: Discovering that some hardware component is at the end of life, so a hardware revision is needed. The hardware has to be shrunk in size to make room for new features such as the camera. A lower cost or lower power consumption part replaces a few discrete components. Some of these changes don't affect the software but plenty of them do. Yesterday's latest technological breakthrough and state of the art is today's obsolete technology.

james@grenning.net

Not only do the features and hardware platform evolve but the third party software we use, to help our products get to market fast, is a moving target as well. New capabilities are added to the Real Time Operating System (RTOS). The Commercial Off-The-Shelf (COTS) packages such as the file system, USB driver, camera driver, and MP3 player all have new capabilities added as well, leading to more change and integration into our system.

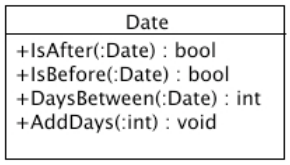
As soft as software is, it is very hard to change. Software must be designed to change. I do not mean that hooks must be in place for all the possible features, but rather that care is required to keep designs clean and flexible so they can effectively evolve as time and technology march on.

Designing for Evolution

I will briefly describe some of the concepts in Object Oriented Design that help keep designs flexible and ready for change. Complex systems are built from smaller less complex pieces. This is the concept of Separation of Concerns (SoC), which aids in partitioning software modules into manageable pieces. Determining the right pieces is a difficult process requiring creativity, experience, and technical knowledge.

Encapsulation

Encapsulation also referred to as information hiding, is about restricting access to the inner workings of a module, so the inner workings are free to change as requirements change, or advances in technology provide better solutions. Think of Y2K and the impact of changing from 2 digit dates to four digit dates. If the date had been hidden in an object that had comparison operations and differencing operations built into it, the number of date digits could have been easily changed. Through encapsulation we hide internal details so that clients of the object cannot depend on the inner workings of the object allowing, the object to be more easily changed as requirements evolve.




```
Date legalAge("2/2/2010");
Date today = Date.Today();
if (today.isAfter(legalAge))
    //your turn to buy
```

Abstraction

Abstraction is the filtering out of details to focus on a more general idea. An abstraction is a name. We practice abstraction in our everyday lives. I could abstract the following process into the concept "drive to work"

1. Start car, back out of garage
2. Turn left out of driveway, turn left at the stop sign
3. Etc…
4. Park the car, turn off the car, walk to the office, unlock the office, go in

Certainly we have to do all those minute steps, but it is really valuable to be able to refer to them abstractly as "drive to work".

When we abstract something we have to name it well. Function names in C are abstractions, module names are abstractions and the operations on modules are also abstractions.

Interface
An interface is the communication boundary between software modules. More flexible systems can be created if modules interact through interfaces, rather than having free-for-all access of common data as is too common in C programs. Interfaces, abstraction and encapsulation go hand in hand.

We experience real world interfaces everyday. Driving your car, you do not have to be concerned with the fuel being used when you press the "gas pedal". A better name for that abstraction is the accelerator. Because of the accelerator interface, I can get in a car anywhere in the world and drive it. When I hop in a rental car I wish the radio and climate control system were as familiar an interface as making the car accelerate, stop, and turn.

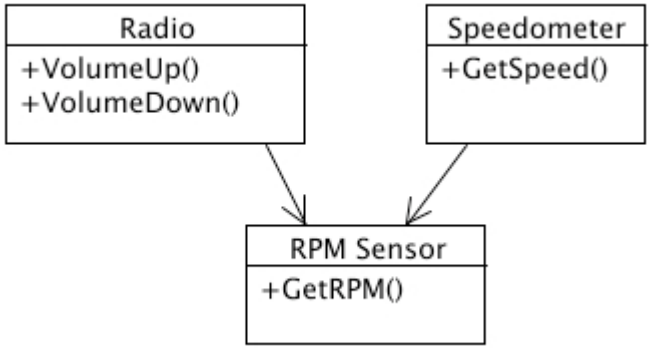
When doing OO design we have objects that interact through interfaces just like I interact with a car through the driver interface. Programming to interfaces means that as long as implementations have the same interface, they are substitutable

Managing Dependencies

With that brief overview and those few simple ideas introduced, let's talk about what happens to software over time. It often starts out clean with a fine architecture, but as changes hammer the code the clean structure decomposes into a mess. It is not good enough to design it right in the beginning, and hope for the best, a design has to evolve with changing requirements. There is no amount of up-front design we can do accommodate for all coming features. And if we try to get all the requirements, we may get stuck in analysis paralysis.

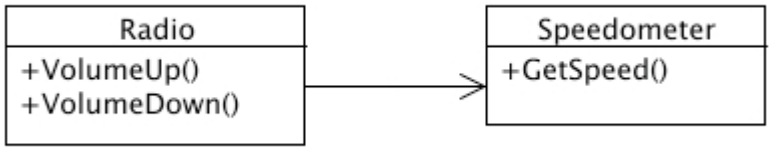
Don't Repeat Yourself
Let's consider this requirements change. The radio that was independent from the whole car in the 2006 model year is required in the 2007 model year to adjust the volume based on the speed of the vehicle. I guess this helps make up for the car being too noisy at higher speeds. Does the radio need to know the RPM of the wheels, does it ask the speedometer for the current speed, or does the speedometer know how to adjust the volume? It's just these kinds of functional dependencies that start the degradation of nice independent designs.

Radio and Speedometer both know the RPM Sensor



This design has the problem of two places having to calculate the speed of the car. This violates the don't repeat yourself principle (DRY)[THOMAS] Duplicate code is a hidden dependency. When a requirement changes, these hidden dependencies have to be tracked down and changed. Cut and paste may seem like a quick way to get the project done, but should be considered a missed opportunity for abstraction.

Radio asks the Speedometer

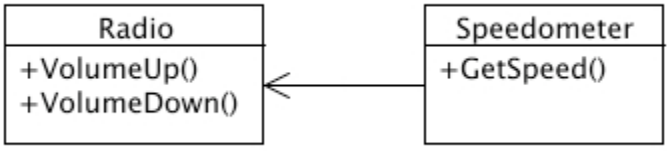


Having the radio ask the speedometer is an improvement, as there is only one speed calculation now. With this design the Radio will have to keep asking.

Tell Don't Ask

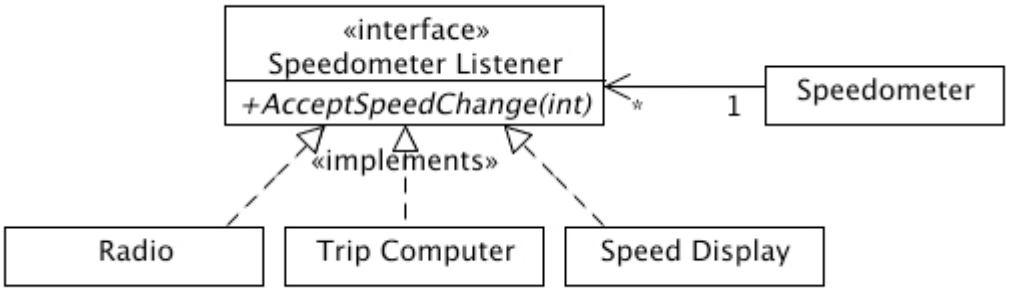
Having the Speedometer tell the Radio is a slight improvement as it knows when the speed changes. This also follows the "tell don't ask" principle. [THOMAS] Too often the choice is made on what is the easiest thing to do now, rather than choosing a change that keeps the design clean and flexible.

Speedometer tells the Radio



This seems odd that speedometers need to know about the Radio. If we follow this model they would have to know about the speed display and the "distance to empty" display and the trip computers "estimated time to arrive" function. It would be better for speedometers to be independent as in this design.

Speedometer is decoupled from listeners



This design has a very important positive attribute; the speedometer is open for extension for new kinds of SpeedometerListeners. The Speedometer will work with any listener, even ones not yet anticipated. This design adheres to the Open/Closed Principle. [MARTIN]

Critical Dependencies to Manage in Embedded Software

The engineering reason to use objects is to manage dependencies. In embedded software development it is especially important to manage dependencies on hardware, RTOS, and software created outside your groups immediate control.

Hardware Dependencies

The importance of managing hardware dependencies hit me back in 2000. I practiced hardware abstraction prior to my ah ha! experience but not with as sound a business reason. I was working with a large communications equipment provider. Their existing and very successful system was based on Time Division Multiplexing, a former state-of-the-art technology. As it happens in our fast paced world the previous state-of-the-art technology was becoming obsolete; TDM was being replaced with Voice Over IP (VOIP).

We were working on implementing some of the use cases for the system and sometimes we needed some very detailed descriptions on the sequence of events and the reaction to those events. The systems engineer for the project would go back to the TDM-product's working C code that he helped develop 10-15 years earlier to find the answers to our questions on required system behavior. After a few occurrences of mining for system behavior, it hit me. If the original designers had worked to keep system behavior separate from hardware specific implementation details, our job would have been much simpler. We could have preserved much of the system behavior and replaced only the hardware specific implementation. No one in 1985 could anticipate specifically that VOIP would be the new innovation to reduce the costs of communications, but certainly the designers could agree that there would be new innovation of some form, making the TDM implementation obsolete.

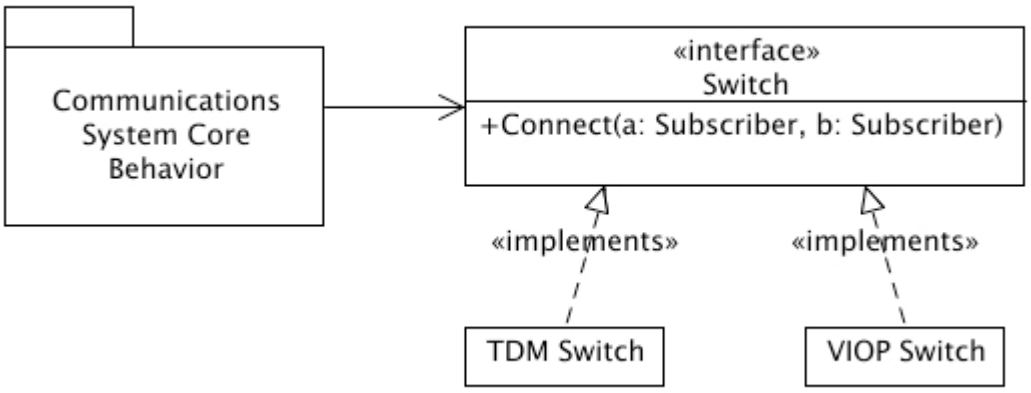
With the benefit of hindsight, this is very easy for me to say. If the concept of Separation of Concerns had been applied, the design would have been much more ready for change.

SoC is very difficult to do with the tools available in the 80's. C is not a good language for separating concerns, it is possible, but the language does not provide much help, and that means specific techniques and discipline is needed. C does not support the "Linguistic Modular Units principle".[MEYER] This principle means that modules should be separately compile-able and that the language supports the concept of a module. C has no concept of a module. Modules can be created through convention only. C++ and Java classes could be considered modules by this principle.

Core Coupled to Hardware Implementation Details

In TDM based communication system, TDM-specific details were intertwined with the application rules defining when one party could communicate to another party. The core of the system managed digital streams and timeslots rather than the more abstract notion of connecting two subscribers. The core business rules were coded with specific knowledge of the hardware implementation and the design became more rigid. The core should have been written to ask the hardware dependent layer for services needed, such as connect subscriber A to subscriber B. These business rules did not change for the hardware evolution from TDM to VOIP. The "why to connect A to B" and the "when to connect A to B" are guided by business rules, while the "how to connect" is hardware specific.

Core Interacting with Abstract Hardware Through and Interface



Here the core behaviors depend on an interface to do the detailed work. There will be plenty of complexity below the switch interface, but the core application will be independent of the hardware implementation, preserving the core behavior investment.

This core behavior is often the place where a company's long-term intellectual property is encoded.

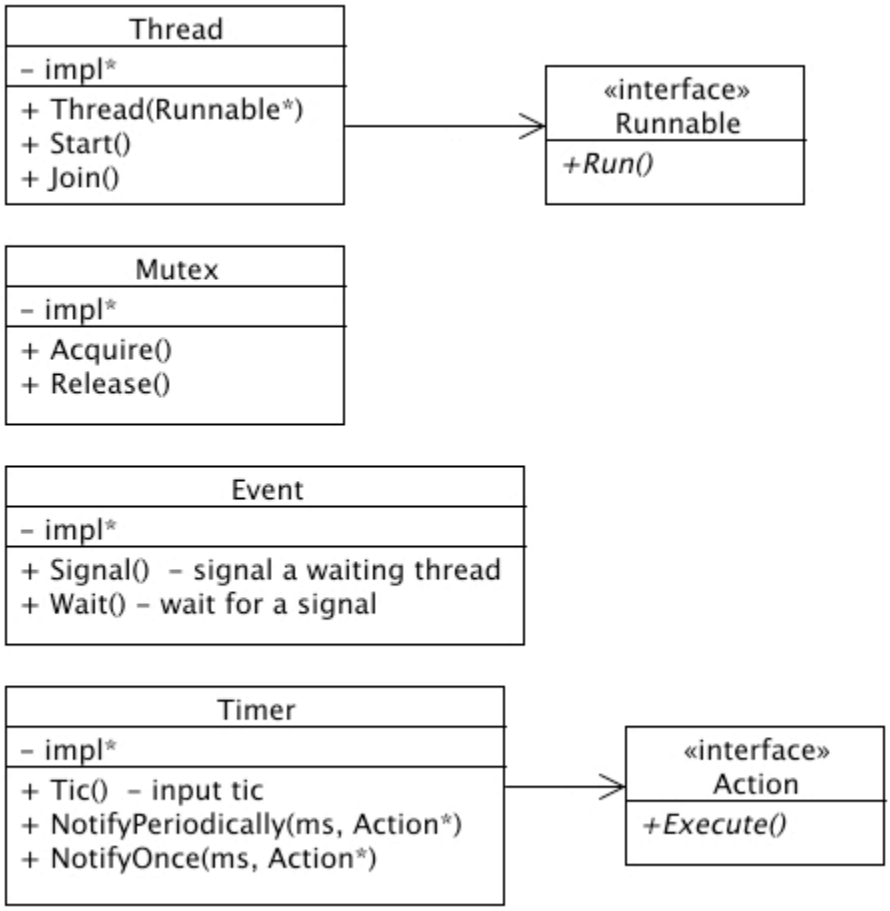
RTOS Dependencies

Depending on a Real Time Operating System (RTOS) is also a problem. Often the APIs are difficult to learn and use, vendors may change the API to introduce new features, causing ripple effects through existing code. There may be reasons to switch to a new RTOS vendor such as lower royalty fees, compatibility with a different processor, or additional capabilities not offered by the current vendor. Intertwining your application with RTOS calls hurts portability. Event if you think you won't need portability, you probably will one day.

Wrapping the RTOS in a very thin OO wrapper can reduce the coupling to the RTOS, simplify its use, and reduce the learning curve for some team members. Wrapping the RTOS can also encourage using a smaller set of concurrency abstractions, cutting down on the conceptual weight of the design. By making your code depend on your RTOS wrapper, it will be easier to change the RTOS or support different RTOS platforms in the future.

One example of an RTOS wrapper is a library that abstracts concurrency responsibilities. CppTestTools is an open source unit and acceptance testing framework developed by Object Mentor that can serve as an example. Its concurrency library is a very simple, but sufficient for its needs, example of a thin OS wrapper.[CPPTEST] There are two supported platforms, one based on gcc and POSIX and the other based on win32. There are unit tests for all the wrapper classes to help assure that each concurrency implementation has the same behavior. The Concurrency library has classes for Thread, Mutex, Event, Timer, and ActiveObject.

CppTestTools Concurrency Classes Providing RTOS Independence

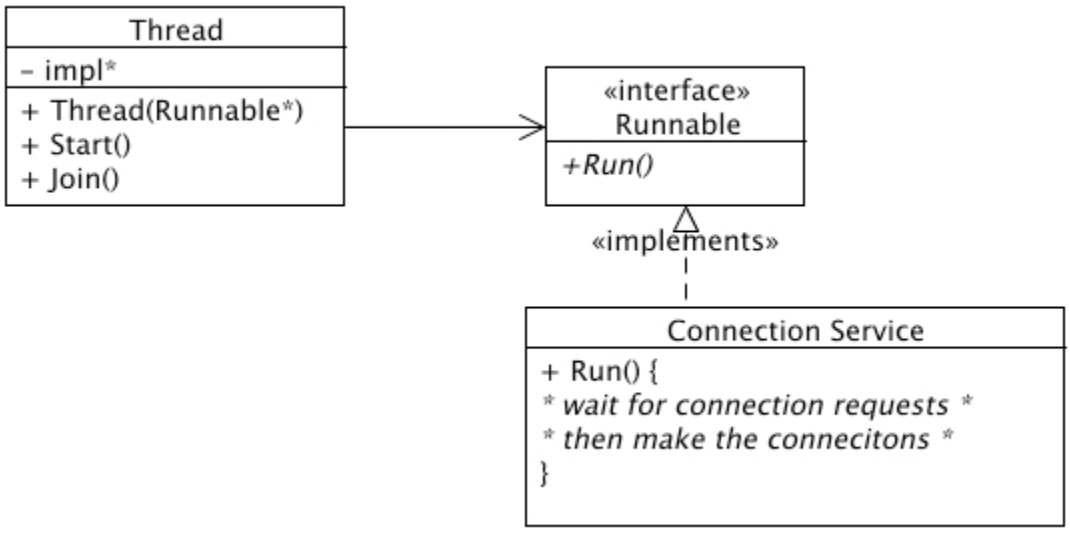


The *impl* element in the private area of each class points to an RTOS specific data structure. The operations on each class translate to the native RTOS specific functions. This design adheres to the Single Responsibility Principle.[MARTIN] Each class encapsulates a single concurrency mechanism. If the need arises to change to another RTOS, only these classes will need modification. If portability between RTOSs is needed, multiple implementations of these mechanisms can be created. Wrapping the RTOS's native mechanisms means that fewer people on the team have to be concerned with exactly how to get the RTOS to behave.

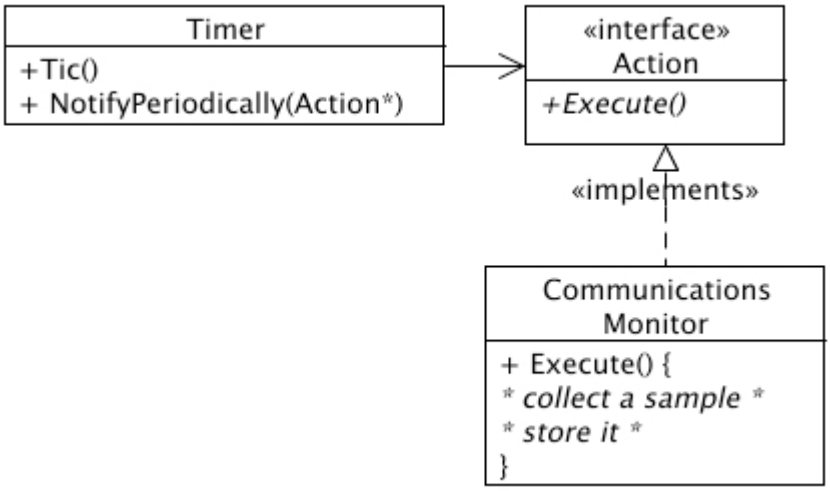
The *Runnable* and *Action* interfaces are used as hooks to call back into application classes that need to interact with the concurrency mechanisms. Using mechanisms like these isolates the threading mechanics. This supports separation of threading logic from application code providing more flexibility in adjusting the concurrency policies as they change over time.

In this example the connection service is run in its own thread illustrating Separation of Concerns. The Thread is created and fed the Connection Service at construction time. The Connection Service's Run method is called in the separate thread through the Runnable interface.

Connection Service Runs in its Own Thread

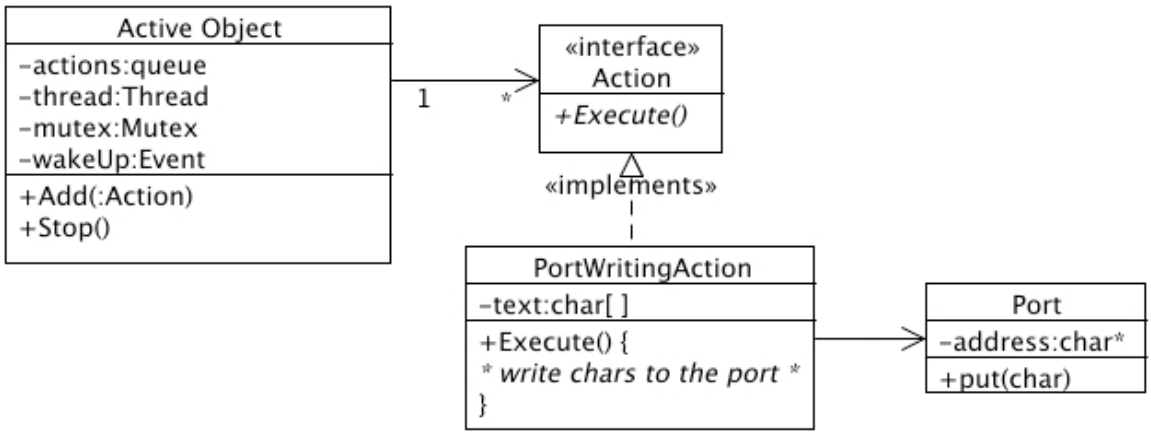


In this example the Communications Monitor is called periodically, through its Action interface, to collect statistics.



From the basic concurrency primitives a very useful concurrency mechanism can be created called an ActiveObject.[SCHMIDT] The cost of creating a Thread may is usually rather expensive, and that cost can be paid once and used over and over again. An ActiveObject has a Thread that runs Actions. A client of an ActiveObject creates an

Action and adds it to the ActiveObject. The ActiveObject's Thread wakes up and runs the Action. Actions are typically queued using a first-in first-out algorithm, but other policies could be implemented. An ActiveObject can be used to queue access to some specific shared resource like a serial port. Very light-weight and flexible concurrency policies can be built from these building blocks.



External Software Dependencies

Dependencies must be managed on software components created outside the group using your components. The components may be commercial off-the-shelf or a component created by some other group in the same company.

I'll use the word component in this discussion to mean any software package whose evolution is out of the client development team's control.

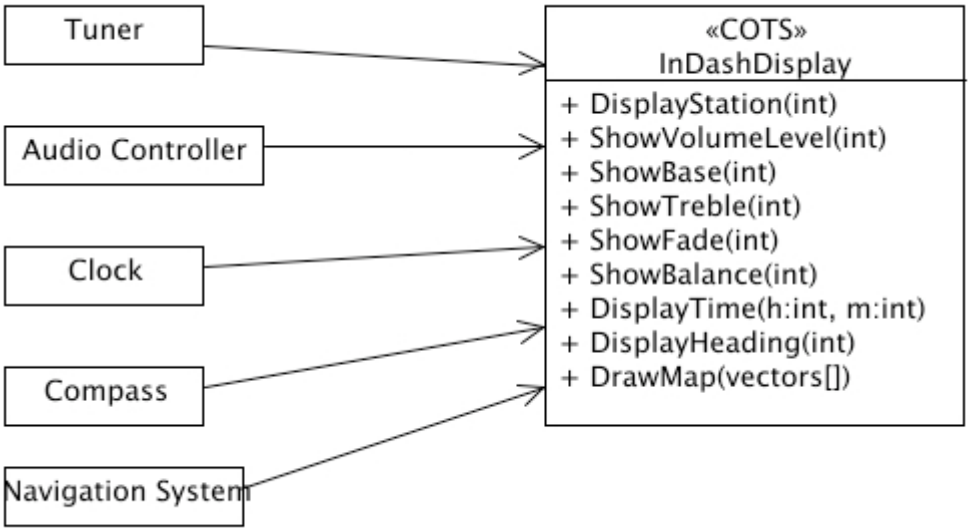
In the case of COTS components, the component providers want their products to be full featured and applicable in many applications. Users of COTS software often only need part of the COTS package's capabilities, and want quality and stability. Unlike internally developed software, COTS software may be changed for reasons that are irrelevant to your product. COTS suppliers enhance and evolve their products based on their business needs, and these changes can make for considerable work in accepting new releases. Another reason to manage dependencies on COTS software is to make it easier to change vendors should a different supplier provide improvements in features, quality or price.

COTS components will likely have a fat API so the package can be all things to all potential users. A specific user wants a simple focused interface, tailored to their specific needs. The component provider and the component users are at odds.

A component provider wants all users to use the latest version, or very few versions at least. They don't want to be burdened with supporting too many releases. The component user wants a stable and reliable version. They don't want to have to keep upgrading and retesting the new releases that may change for reasons the user does not care about. Again, the component provider and the component user are at odds.

Managing the dependency on external components and on the releases of the components can be done by applying The Interface Segregation Principle. ISP states that clients should depend on interfaces that are tailored to their needs, not the needs of the server.[MARTIN] Interfaces exist for the benefit of the client. The component designers create fat interfaces, but component users can manage the dependency by creating focused interfaces.

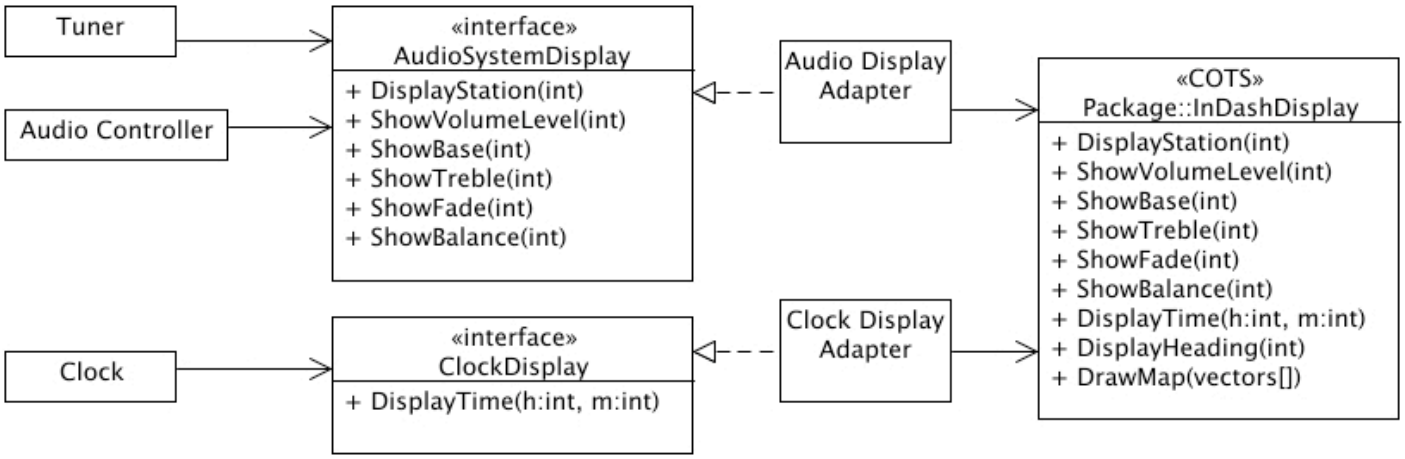
Interface Segregation Principle Violation with COTS



The InDashDisplay is made by your Taiwan office. The clients of InDashDisplay are distributed across the world and InDashDisplay continues to evolve causing considerable churn in the source code of the users of the InDashDisplay.

Having the Tuner, Audio Controller, Clock, Compass, and Navigation System define their own interfaces, and then adapt them to the provided interface can easily solve the problem of a fat interface, allowing the different client teams more freedom during development.

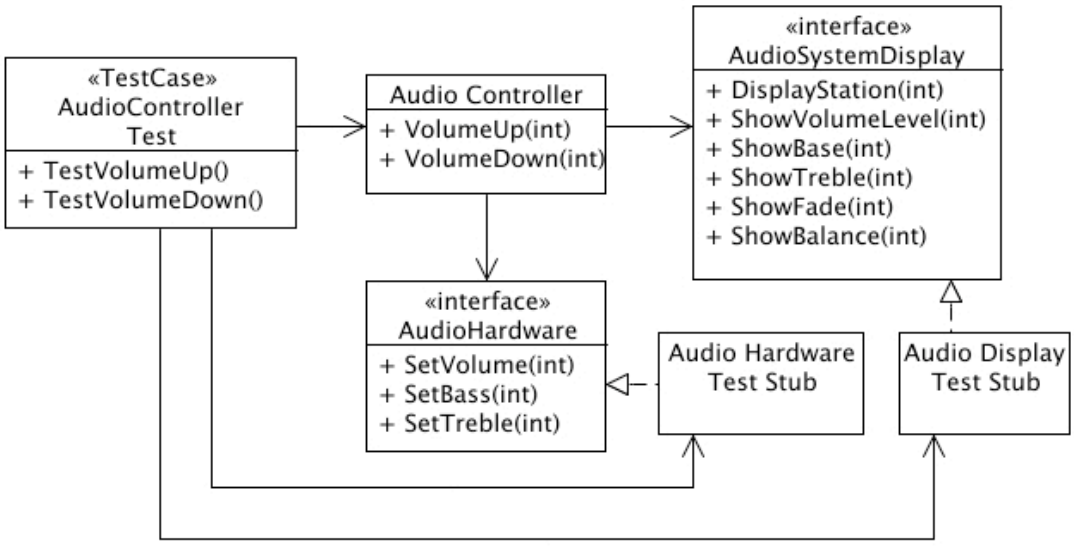
Interface Segregation Principle Applied



Here the client team defines interfaces and small adapter is written to convert from the perfect interface to the provided interface. This design has an added benefit that it shows exactly what capabilities of the external component are being used and how. When changes occur to the external component's API only the adapters have to be updated.

Design For Test

By applying these design techniques the embedded software becomes more testable, as well. Abstracting the hardware enables test drivers, stubs and fakes to be put in place to help exercise the software. By staying independent of the RTOS, tests could be run in the development machine allowing testing to begin before the target is ready and to make up for scarcity of target systems. By having client software define their ideal interfaces along with adapters, test stubs again can replace real components to facilitate testing subsystems independently. A team could also invest in automated acceptance tests to confirm component compatibility when versions change. The client defined interface narrows down what must be tested in the third party component by acting as a specification of how the application expects to use the interface.



When an application module, like AudioController, is decoupled from the hardware and from other components, it can be tested in a cost effective and repeatable way. Audio Controller Test is a test case that is compiled and run to check that the Audio Controller behaves properly. The test case simulates the events coming from the hardware, such as a change in the volume control. When the Audio Controller receives the volume up event, it will instruct the hardware to adjust the output level, and instruct the display to show the change in audio level. The test case checks the test stubs to see if they were given the right instructions. Test cases would also be developed for interesting boundary conditions, like maximum volume, and minimum volume.

These tests are cost effective in that expensive external automation could be avoided for more exhaustive behavioral tests in simulation. Tests can be batched together and run without manual intervention eliminating some manual test effort. The automation makes the tests inherently repeatable, removing the human element from the test execution. See my other conference paper on Test Driven development.[GREN-TDD]

Cost of Good Design

What about the performance overhead imposed by these design techniques. Do not pre-optimize. Write well structured programs, measure and optimize where needed based on data.[NEWCOMER] Another respected computer scientist says Make it work, make it right, make it fast.[JOHNSON] I would add to the esteemed Mr. Johnson's advice to make it fast enough. For embedded systems we also have to be concerned with small enough as well. I think this means budgeting memory usage and tracking its usage through the life of the project and product.

Ok, that's fine, but what do objects cost? This question has different answers depending on language. Most readers of this article are probably interested in the C to C++ comparison. C++ is C, so many of the performance characteristics are the same. There is a cost for the extra capabilities in C++. Bjarne Stroustrup did not want to penalize

developers that did not some OO feature. Take a look at my paper "Why are you still using C?" for a more detailed discussion on C vs. C++. [GRENNING]

Here are a few thoughts on the cost of OO in C++.

- When invoking a method on an object there is an implied first parameter to each function call, which is the pointer to the object being worked on. It is the same overhead as passing struct pointer to a function.
- Invoking a static class method is the same as a C function
- Objects can be passed and returned by value and this can be very expensive in stack space and time. Prevent passing/returning of objects by value by declaring the copy constructor and assignment operator in the private area of the class, and then do not implement them
- Exceptions have unspecified space and performance characteristics. My quick advice is to disable exceptions until you know them and your compiler better.

Polymorphism is the mechanism used when calling through an interface as in SpeedometerListener. Polymorphism has a cost, as does the alternative non-OO approach using switch/case for selecting the alternative implementations.

What is the cost of polymorphism?

- Each class that has at least one virtual function has a virtual table (vtbl) in memory
- The vtbl contains a pointer for each virtual function in the class
- Each object that has a virtual method has a hidden data member that points to the virtual table for the object's class
- The code to call a virtual method through the vtbl is bigger than a direct function call.
    - Like **call vptr[offset]**
- Non-OO programs use switch/case statements or if/else statements to choose what function is called for some special case. This has a cost. But comparable to or better than the switch/case alternative

OO Design in C (Only a few words)

The C programming language does not provide a lot of help in designing loosely coupled modules with high cohesion. OO languages such as C++ and Java obey Meyer's Linguistic Modular Units principle, as the language provides support for modularity in the language in the form of classes and namespaces. C programmers have to use conventions and discipline to build modular systems.

The basic mechanisms needed are

- Interfaces
- Encapsulation
- Substitutability

Interfaces, and encapsulation and can be accomplished by using C header files along with their C source files as classes. The header file would provide all the operations on the

module defined as function prototypes. In the simplest form the data for the module would be in the C source file as static data. Data is encapsulated and surrounded by functions that operate on that data. Only these functions are allowed to access the data directly and this is where the discipline must be applied. For substitutability the linker can be used to select different version of the modules, either for testing purposes of for different hardware implementations. If runtime configuration and substitutability is needed, functions pointers can be used. If you are really thinking of doing OO in C, ask yourself why and take a look at my paper "Why are you still using C?". [GRENNING]

Final Comments

Using Object Oriented Design principles can help produce a more flexible system made up of cohesive modules that are loosely coupled. Producing a modular design that embraces Separation of Concerns will help today by promoting a cleaner and easier to test design. Debugging will be easier, too, and also less frequent! It should pay for itself again in the future with easier portability, clearer responsibilities, and looser coupling.

Design is not a one-time activity. It continues through the life of the software, release after release. Change and time to market work to degrade the design like entropy increases the disorder of physical systems. Energy must be put back into the system to maintain order and preserve the value and prolong the useful life of the software.

[THOMAS] Thomas, Dave, Hunt, Andy, The Pragmatic Programmer, Addison-Wesley Oct 1999

[MEYER] Meyer, Bertrand, Object Oriented Software Construction, Second Edition, 1997, Prentice Hall

[CPPTEST] www.fitnesse.org/FitServers.CppFit.CppTestTools

[MARTIN] Martin, Robert C., Principles, Patterns and Practices of Agile Software Development.

[SCHMIDT] Schmidt, Douglas, and Lavender, Greg, Active Object: an Object Behavioral Pattern for Concurrent Programming, Proceedings of the 2nd Pattern Languages of Programs Conference, September 1995

[GREN-TDD] Grenning, James, Test Driven Development for Embedded Software, ESC-241, San Jose 2007

[NEWCOMER] Newcomer, Dr. Joseph M. Optimization: Your Worst Enemy. http://www.flounder.com/optimization.htm

[JOHNSON] Johnson, Ralph, Performance Optimization, 1998 http://st-www.cs.uiuc.edu/users/johnson/cs497/notes98/performance.pdf

[GRENNING] Grenning, James, Why are you still using C?, March 2003, http://www.embedded.com/columns/es/showArticle.jhtml?articleID=9901135 or http://www.objectmentor.com/resources/publishedArticles.html