Source Code modules and layering

What is layering?

For me software layering is one of the fundamental pillars of physical software design. Whether we recognise it all software is layered. In simplest terms layering is the process of building source code on top of source code. A more considered approach recognises the process. Like a brick built house, rows of layers (bricks) are placed on those below, each layer has well defined boundaries; the downward boundary is the dependencies on lower layers, the upward boundary is the interface it expose and the layers above.

The concept of layering is tied up with the concepts of modules because layers are built from modules.



Figure 1 - Simple layering

Mode formally:

- ? A layer is one or more modules of a system which do not depend on one another.
- ? Layering is a physical mechanism for building software modules on top of existing modules.
- ? A layer will depend on layers *below* it but should never depend on a layer *above*. Layering is transitive downwards: if layer 3 depends on 2 which depends on 1, then 3 also depends on 1.
- ? A module is mapped from the logical to the physical world using a namespace in C++ and a package Java. (If your compiler doesn't yet support namespaces or you are using some other language, you can, and should, apply the same concepts but you will need a little more discipline.)
- ? A module is not always a layer but frequently is.
- ? When a module as the smallest piece of reusability layering is important in understanding dependencies.
- ? Layering dependencies are always in a downward direction; an upward dependency opens a circular dependency because of the transitivity.

The lower layers (OS, C library, C++ library) are usually taken for granted: in UNIX systems it can be hard to define the line between the C library is the OS library. These layers represent the foundations upon which we will build our programs.

These modules represent a good model of what we should aim to achieve: namely, they are not dependent on our code and the interfaces are well defined.

Once again the main reference for layering is John Lakos¹. (This cannot be the only book which talks in depth about physical software design but most literature on software design deals with logical rather than physical design.) Lakos likes to number layers, starting at zero for the OS libraries and working upwards. Although this helps with producing a theoretical description of layers in practise it can cause documentation problems because layers are added, removed and modules split.

Modules are abstractions

A module provides an abstract representation of some area of functionality. Like an iceberg, only a small part is visible (the header files, the library file) while most of the module is below the surface in the source code files which are not exposed to the outside world.

This carries over to a layer. Frequently a layer is just one module, lets call it M1. Other modules use this module, they are said to be *layered on top*. However, a layer may contain modules M1, M2 and M3. As long as M1, M2 and M3 are in no way dependent on one another they can co-habit in one layer, say L1. But if M2 and M3 in some way depend on M1 (i.e. they use something in M1) then M2 and M3 are in a layer above M1, now we have L1 containing M1, while the next layer, L2, contains modules M2 and M3.

Within a module and a layer, it is natural to find classes which depend on others. This does not imply more layers because these dependencies are hidden, below the surface, so the users of the module do not need to know about the dependency. (You may still find is useful like to think of internal layers.) Every module exposes an interface. This should be small enough to perform it's intended role without confusing the user. It should be well documented. The module should have some central concept, e.g. database abstraction or inter-process communication. It is important that this central concept be clearly stated and respected by the module. It would not do for a database layer to also contain functions to control a robot-arm. This makes the module confusing and will detract from it's usefulness. One exception here is the idea of a utilities module set out in the first example. Within such a module you may diverse functions such as string manipulation, complex numbers, etc. This is an issue of practicality, the utility module effectively contains a lot of small modules; it is easier to manage one such module than a multitude of small modules. The defining concept is: a collection of utilities. If one area of functionality starts to stand out it is time to split the module in two (or more) which may be independent of each other or layered.

Layers are not modules

A layer may not have a central concept because it can hold several modules which have different central concepts. Each layer is a physical grouping while a module is a logical grouping.

Recognise which layers are above and below your layer. If you don't need to be dependent on a layer then don't be, additional dependencies detracts from flexibility and hence maintainability and re-use. The physical manifestation of a layer may be difficult to see in raw source code because dependencies are not always clear. Lakos' numbering scheme helps enumerate the layers of a system but makes documentation quickly dated, as for source code it would mean identifying which layer every file was in and because layers may be inserted and removed keeping the comments up to date would be impossible.

In my previous article on include files² I suggested you divide you header files between: system, project and local, this is a basic form of layering.

Some examples

Example 1



One system I worked on in the last few years had a layering model which looked a bit like figure 2.

Figure 2 Example layered application

Just about any system we build will have the OS libraries at the bottom. On top of this will come the libraries of our language. As such the lowest two levels of any system are fixed.

Many projects, organisations and even individual developers have a set of common utilities which are used again and again, e.g. IP translation, string conversions, etc. In many cases these are syntactic sugar on some feature the OS or language provides but we choose to wrapper it in some fashion for what ever reason. This utilities layer often becomes as much a part of our foundation as the language libraries.

In this particular system there are two modules sharing one layer: GUI utilities and IPC utilities. These two modules have no dependency upon one another but are both dependent on our general utilities, higher layers depend on both modules although they need not. While we build layer upon layer it is good to only build upon the layers we need to. Suppose instead that the IPC layer depended on the GUI layer: this would make the IPC layer less flexible because it cannot be used without the GUI layer; it also becomes vulnerable to changes in the GUI layer.

The top layer in almost any system is always the same: the application layer. The layer which implements our business application.

I say almost because there are two obvious exceptions:

- We may be building a product which expose an API which is used to build the final application,
 e.g. I'm sure Sybase has many layers of source code below ct-lib, but ct-lib is not an application in itself.
- ? There may be further layers on top of application, e.g. the application may be free of user interface which is added as a *presentation* layer on top, imagine a application which exposes a graphical interface for PCs and a text interface for mainframes.

Downward transitivity between layers is taken as given in a dependency diagram like the one above. Although some may choose to show the application layer depending directly on the utilities this quickly leads to excessive complexity in a diagram. Sometime, as in the next example, you may want a higher layer not to see a lower layer: physically there is a dependency but you can logically hide it through program syntax and design constructs such as abstract interfaces.

Example 2

On another project we faced the challenge of porting an Sybase based application to Oracle and Microsoft SQL Server. The solution was an entire database layer. The interface to this layer was loosely based on the Sybase db-lib interface to simplify the porting. Below this interface the layer allowed drivers for Oracle, Sybase and Microsoft databases to be plugged in.

Once the interface and Sybase driver was complete we where able to "jack up" the application and slip our database layer below the application which was reworked to access the database layer rather than direct to Sybase. At this point the app worked fine, we where able to develop Microsoft and Oracle drivers while the layer was in use.



Figure 3 Lifting an application to insert a new layer

With this example I am deliberately trying to draw parallels to the way buildings are constructed – the standard libraries are the foundations, next comes our own concrete base then the floors of our building. The ground floor holds up the first floor and so on.

The layer is good because it abstracts and simplifies. It allows different areas of the system to be worked on at the same time while the interface boundaries are respected. In the above example work continued on the main application while the database layer was being added.

However, abstraction can lead to problems precisely because details are hidden. In the database example all the layering didn't save us from having to re-work the SQL contained in the main application. This still had to be reduced to a common subset for all three databases. Anyone maintaining the system could not forget this, although there was an abstraction in place which made the system portable it still imposed constrains on developers. If these constraints where not public they could lead to problems.

Application and Solution domain

It is becoming increasingly common to talk about an *Application domain* and a *Solution Domain* – see Ian Bruntlett³ and Jim Coplien⁴. The application domain is what used to be called *Business Logic* while the solution domain details the elements that we will use to solve a problem, e.g. Windows 2000 with a Sybase database, or AIX with an Oracle database.

In general the solution domain comprises the lower levels of the system and aims to abstract some of the complexities away allowing us to develop the business logic free from considerations of OS quirks, or database.

In truth it's not quite this simple. We can abstract ourselves away from the underlying technologies without introducing inflexibility. In addition, there is aways the possibility of an *Outside Context Problem*⁵ occurring: the above database example would be rendered useless if faced with a request to port to an Object-Oriented database such as Versant.

Packaging a module

The module interface should be provided by one or, more usually, several header files. As these are related to a namespace place them in a separate, publicly accessible, directory of their own⁶. Most importantly the module must be presented for use by packaging it. Typically this is done with a static library (.lib) or, a shared/dynamic library (.so in UNIX/.dll in Windows). The module forms a collection of files which are compiled and packaged together.

Personally, I prefer static link libraries to dynamic libraries. For the following reasons:

- ? Although dynamic libraries once represented a way to save memory, this is usually less of a consideration today.
- ? Static linking ensures we have no undefined symbols, or mis-matches at compile time not run time.
- ? Start up times for applications with many dynamic libraries to load can be longer than a "fat" binary.
- ? With a dynamic library it's possible the wrong one will be loaded (most Windows programmers have suffered this at some time).
- ? Dynamic linking means we need to distribute and version control a separate customer deliverable. (This may be an advantage and a powerful way to cope with some variances.)

The module and hence the layer should have clearly defined dependencies, you should be able to diagram these dependencies in an acyclic graph. If module X depends on Y, which depends on Z which depends on X you have a problem. A change to X may ripple to Y, which may ripple to Z, which.....

Perhaps one of the greatest advantages of dividing you system into modules and layers is that you form firewalls between these layers where such dependency problems will become obvious.

It is not uncommon for a module to be applicable to more than one project, knowing what the dependencies are increases the chances it will work in the next project as is.

Conclusion

The most important thing about modules and layers is to recognise where they exist. Decide what your modules and layers are early in your physical design, don't be afraid to add new modules and layers, and remove others: *layer early - layer often*. If you can continue to add (and remove) modules you are demonstrating that your system has flexibility.

What you should avoid at all times is cyclic dependencies. Although a linker may pick this up it may not, and what one linker accepts may be rejected by another. If you have lots of dynamic libraries you may not see the problem at all.

When devising modules and mapping layers my guiding principals is: a place for everything, and everything in it's place.

¹ Large Scale C++ Software Design, John Lakos, Addison-Wesley, 1996

² Overload 39

³ Overload 39, September 2000.

⁴ Multi-Paradigm design in C++, Jim Coplien, 2000.

⁵ Thanks to Ian Bruntlett (Overload 39) for describing how the *Outside Context Problem* of Iain M. Banks *Excession* novel (1996) is applicable to the software world.

⁶ Kevlin Henney, Overload 39.