Software Design and Architectures
SE-2 / SE426 / CS446 / ECE426

Lecture 4 : Layered Systems

Layering is probably the most fundamental architectural pattern, used by all large software systems.

What we must do now is to begin to look at how software systems are organized at the 'highest level', which will mean at the level most clearly approachable by a new member of the team. Although the final architectural form of a system may not be and should not be fixed too early, nevertheless a certain feeling of clarity, of *aha!* appears in the design and the team's understanding, as the architecture (in particular) takes firm shape. Holt calls this the 'shared mental model' phenomenon.

Buschmann says *the Layers patterns describes the most widespread principle of architectural subdivision... many of the block diagrams we see in system architecture documents seem to imply a layered architecture.*

Example : Unix Kernel
This diagram models the basic structure below the I/O entry points in the Unix kernel. Historically the sockets/network part of this was added later, and it fit 'naturally' into the architecture because the new material was below the kernel entry point. Similarly, Unix has been ported to many different hardwares, and new device drivers are easily added, again because these lower levels don't depend on the higher levels.

By contrast some operating systems (e.g. DOS) had dependencies in both directions, as when the BIOS talked directly to the screen to handle an error (Abort/Restart/Retry)...

[BSD Unix Layered Architecture]

Example: NextGen Layered Architecture ([Larman])
This sample development is a point-of-sale system which is the subject of Larman's book *Applying UML and Patterns*. Not all the packages are shown. (These are logical packages not development ones.) Note the dependency, again downward, notupward. The application logic (*e.g.* pricing strategy) is not dependent on the user interface: and so it can change independently, and new interfaces can be added without affecting the application proper.

Note that there a layer is quite a large structuring with relationships within as well as downward. We don't see any upward dependencies, though.

[NextGen POS System]

Example: TCP/IP protocol stack
[Buschmann p 44] The most widely used communication protocol, TCP/IP, consists of four layers, of which the first and the fourth are variable. This one is typical. Each layer is characterized not by the details of the functional interface (in terms of entry point/subroutine signatures) which vary from vendor to vendor, but very strictly in terms of the *data format* and the *state transition behaviour* of each layer. This is exactly so that the software supporting a given layer can be distributed across multiple network nodes and even be provided by different vendors and be completely different software. Buschmann calls

In this diagram the looped arrows are protocols and the downward arrows are a procedure calls on a given node. The downward arrows are often less strictly layered than the diagram suggests: a vendor may provide all the software, in the stack. on a

(Note use of term 'stack' as a jargon equivalent for 'layered system', especially when layers are viewed as pretty plug-replaceable. Terminology from networking, has been expanded.)

[FTP/TCP/IP Layers]

Example: PostgreSQL in layers
The large software system is invariably layered. So much so that the arrows are usually omitted, with the implication that dependency and use point downward, never upward.

[CBMS Reference Architecture]

[PostgreSQL Recovered Architecture [

Example: Application/JFC Swing/Java Platform/Cocoa/Darwin
The implementation of a modern Java application using javax.swing, running on a layered operating system, shows *separation of concerns*, especially to the degree provided by the virtual machine layer, which enables the upper layers to run on any other operating system where the JVM has been implemented.

In this case each layer has a well-defined interface and can be versioned, installed, and maintained independently of the other layers.

[Swing Application Diagram]

The Layers Pattern
In each of these situations we have a large or huge system in which variation of parts is expected, and in which there is a very wide range of functional roles to be played and functionality to provide. The dominant characteristic is a mix of low- and high-level issues [Buschmann p 32]. Examples of low-level issues: hardware traps, sensor input, bits on a device, signals from a wire. Examples of high-end issues: multi-user distributed interface; telephony billing tarrifs and rules. Really this is true of any large software system. Typical is the movement of 'events' from high in the stack, downward, with (as we shall see) callbacks to move information and control back up. And the following problems are always prowling outside the door of any such system:

- changes ripple through the system, tracking the flow of information and control
- application logic (”business rules”) and interface logic get mixed up, making evolution prohibitively expensive
- general technical (low-level) services get mixed up, preventing their reuse or replacement
- unrelated areas of concern are linked (“coupled”) making it hard to assign tasks to specialists
- testing changes seems to affect the whole thing, making bug finding (“assigning blame”) difficult

[Typical Generic Layers diagram]

Assigning Responsibilities to Layers
The keys to a successful layered architecture are the appropriate assignment of responsibilities to each layer, and a high degree of independence between layers, *especially* in the upward direction. The task of the highest layer is the overall system task as perceived by clients or users. The other layers' tasks are to help layers above by shouldering some ‘lower-level’ responsibilities. It's hard to do in general, but skill and experience (just like in building architecture) suggests *reference architectures* for commonly occuring situations.

We saw the DBMS reference architecture earlier (without particularly careful assignment of responsibilties.)

This is Craig Larman's assignment of responsibilities to layers in a typical information system. The separation of concerns at the UI level is particularly noticeable, with Presentation, Application, and Domain layers, which will later be studied as a UI design pattern.

[Larman IS Layers]

This is the OSI 7-layer Network Reference Model. It is a (standard) assignment of responsibilities to layers of a network implementation.

[OSI Model]

In skillful practice there will be an interface or collection of interfaces which define access to each layer *from above*. This may not always be necessary but it helps when replacing or changing the design of a layer (while preserving its responsibiitliies.) Shortly we'll look at the scenario in which information and control rise from below. There will also be structure *within* the layer, which to its less senior developers may seem like the whole world! Later we will consider both the internal 'subarchitectures' of layered systems, and skillful design practices ("patterns") which help to structure layers.

Buschmann adds a hypothetical reference model for strategy games: layer 1, elementary units (bishop, tank); layer 2: basic moves (castle, attach ahead, fire); layer 3: local tactics (Sicilian defense); layer 4: overall strategies. This may help but I think that

Communication Between Layers
The typical scenario is that in which a client issues a request, at the topmost layer. It it translated at each layer into possible many operations at the layer below, until the bottom layer has handled everything. Control or control abstractions are passed through the interfaces. Replies may be handled on the way back up.

A second typical scenario is that in which a hardware event is detected by the bottommost layer. It is internalized and passed to layer 2, which interprets it possibly as part of a more complex event description which eventually is passed on further up.

The passing down scenario is often called a "request" and the passing up scenario a "notification". [Buschmann p 36]. A decision to use only requests is called a *push* model; conversely using only notifications is a *pull* model.

It may happen that a request or notification can be handled without going all the way down (or up). For example, a cache may be kept at a middle layer. Or, a request may have related only to the state of the middle layer. Or, a notification may be ignorable (e.g. *resend* from below after a message has already been sent.)

Error Handling

It's essential to have an error handling plan, which amounts to an 'unepexected' change in direction of a request/notification. Errors have to be translated as well, to prevent users seeing low-level errors. Problem: crosscutting concern.

References
Buschmann ch 2, Beginning through "Layers"
Larman sec 30.2 p 450ff
A Fowler, "A Swing Architecture Overview", at

http://java.sun.com/products/jfc/tsc/articles/architecture/index.html
Venners, "Inside of the Java Virtual Machine", Chapter 2, at http://www.artima.com/insidejvm/ed2/platindep.html
D Reilly, "Inside Java, the Java Virtual Machine", at http://www.javacoffeebreak.com/articles/inside_java/insidejava-jan99.html