COSC345 Week 23 Notes

Commodity, firmness, and delight

"It has been generally assumed that a complete theory of architecture is always concerned essentially in some way or another with these three interrelated terms, which, in Vitruvius' Latin text, are given as firmitas, utilitas, and venustas (*i.e.,* structural stability, appropriate spatial accommodation, and attractive appearance). ... The notion that a building is defective unless the spaces provided are adequate and appropriate for their intended usage would seem obvious. Yet the statement itself has been a source of controversy since the 1960s. The main reasons for the controversy are: first, whereas there are seldom exact statistical means of computing spatial adequacy or appropriateness, there are many building types or building elements for which one cannot even establish the optimum forms and dimensions with any confidence that they will be generally accepted. Second, edifices are frequently used for purposes other than those for which they were originally planned. Furthermore, there is some doubt as to whether 'form follows function' or 'function follows form', since, although, in general, it can reasonably be assumed that an architect's task is to construct specific spaces for the fulfillment of predetermined functions, there is plenty of historical evidence to suggest that many important social institutions have resulted from spaces already built. No better example could be found than the evolution of parliamentary systems. The British system, based on the concept of legislatures in which the sovereign's government and the sovereign's opposition confront each other, originated in the fact that the earliest parliaments met in the medieval palace chapel. The French system, created concurrently with the Greek and Roman revivals, was based on the concept of legislatures addressed by orators, and its environment was that of an antique theatre. In the former system the seating was designed in accordance with the liturgical requirements of a Christian church; in the latter, with the evolution of Greek drama. Neither had anything to do with preconceived notions regarding the most effective environment for parliamentary debate, yet both have had divergent influences on constitutional procedures, thereby deeply affecting the whole theory of government." — Encyclopaedia Britannica

The main ideas in this lecture are

1. Software can *have* an architecture, a large scale structure that is something other than a mere list (even a hierarchical list) of modules
2. The architecture can exist before the program does
3. Architectures are usually expressed in diagrams
4. It is possible to determine some of the properties of a system by inspecting architecture diagrams

From the TOGAF white paper:

"What is Architecture in the context of TOGAF?

ISO/IEC 42010:2007 defines 'architecture' as:

'The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.'

TOGAF embraces and extends this definition. In TOGAF, 'architecture' has two meanings depending upon the context:

1. A formal description of a system, or a detailed plan of the system at a component level to guide its implementation.

2. The structure of components, their inter-relationships, and the principles and guidelines governing their design and evolution over time." — TOGAF Version 9.1 Enterprise Edition, An Introduction; A White Paper by: Andrew Josey, The Open Group, December, 2011, page 6.

The "Bath" reference on the first slide may be obscure. I had long heard about the wonderful architecture of some of the buildings in Bath in England. The first time I visited Bath, I took a tour which took us around some of the sights of Bath. That's when I saw the *backs* of some of the famous buildings as well as the fronts. It turns out that the famous architectures of those days designed *only* the façade of the famously beautiful terraces. Units were built by different builders, and no two were alike. Around the back, units don't have the same number of windows. They don't even have the same *levels*. This is in contrast to modern architecture, where the architects are supposed to design down to quite a detailed level, and we expect the *whole* design to be done by the architect. You could find your own example, adapt this one, or just plainly make the point that

"Software architectures are supposed to describe the whole of some software artefact down to an *appropriate* level of detail." — me

"Appropriate" depends on context, of course. The highest level will typically show the interactions between the system and its environment.

There are architectural design patterns. For example, the Gang of Five patterns book lists a number of architectural patterns, from things like the "Three-Tier" pattern (see for example http://www.linuxjournal.com/article/3508 if that's not familiar to you) to things like "Pipes and Filters" (the Unix Way).

Layering is one of the classic approaches. For example, people using Visual Basic these days (but not VBA) will have this series of layers:

1. Hardware (x86, x86-64, Itanium)
2. BIOS/HAL
3. Windows kernel
4. Windows DLLs and services
5. .NET
6. Common Language Runtime
7. Visual Basic.NET
8. their own code

There's one thing that a well drawn layer diagram lets you see at a glance: how many of the layers do I actually have to deal with? The answer in the case of VB.Net is that pretty much any kind of object implemented in any of several languages (C#, F#, VB.Net, ...) could become visible to a VB.Net program, and you had better be aware of quite a lot about the .NET platform.

The pipes-and-filters technique is a simple kind of data flow approach. For something more general, see http://en.wikipedia.org/wiki/Pipeline_(software) and specifically the link to http://en.wikipedia.org/wiki/Hartmann_pipeline

"Pipeline programming involves applying 'pipethink' to break a problem into a number of small steps, each of which can then be performed by a simple program. Wherever possible, a pipeline programmer uses existing programs as the stages in a pipeline. Traditionally, programs that run in pipelines are small and have one very well-defined function, but they should also be as general-purpose as possible, to allow re-use. Because they are so small and well-defined, it is possible to make them very reliable." — Melinda Varian

This is also a form of *component-based programming*.

The repository model is very close to the way people use SQL data bases, except that the repository need not be a relational data base and need not be a disc-based data base of any kind. The central ideas in the repository model are the clear separation between (the structure and meaning of the data) and (processing the data) and that the whole state of the system is supposed to be captured in the data, not in ephemeral interactions between the components that plug into it. It's relatively common for data to outlive not only programs but operating systems, so when that's the case, pulling the data out of the programs so that you can keep it safely for a long time is a good idea.

When we are looking at architecture diagrams, we can ask questions about "how much" (data), "how many" (transactions per minute), "how secure" (is this supposed to be, and what makes sure that it is). What we're particularly looking for is omissions, things which the system is supposed to do but which we've made no provision for. Some things to look for include logging, security, capacity, redundancy (if a system is supposed to be fault-tolerant it must have some redundancy; other kinds may be less good), distribution, capacity of channels, ...

What I generally do is take a system I'm correctly working on or using and draw a diagram for it and point things out.

In the lecture I intend to draw some diagrams. The point is not to *have* the diagrams, if that were the case I'd give you handouts with the finished version. The point is to *develop* the diagrams, to illustrate the process.

Ideas from *Microsoft Application Architecture Guide, 2nd Edition*.

• **Build to change instead of building to last.** Consider how the application may need to change over time to address new requirements and challenges, and build in the flexibility to support this.

• **Model to analyze and reduce risk.** Use design tools, modeling systems such as Unified Modeling Language (UML), and visualizations where appropriate to help you capture requirements and architectural and design decisions, and to analyze their impact. However, do not formalize the model to the extent that it suppresses the capability to iterate and adapt the design easily.

• **Use models and visualizations as a communication and collaboration tool.** Efficient communication of the design, the decisions you make, and ongoing changes to the design, is critical to good architecture. Use models, views, and other visualizations of the architecture to communicate and share your design efficiently with all the stakeholders, and to enable rapid communication of changes to the design.

• **Identify key engineering decisions.** Use the information in this guide to understand the key engineering decisions and the areas where mistakes are most often made. Invest in getting these key decisions right the first time so that the design is more flexible and less likely to be broken by changes.

While you can buy that book in hard copy, you may find the web version at http://msdn.microsoft.com/en-us/library/ff650706.aspx more convenient.