## Software Design and Architectures
## CS446 / ECE452 / CS646 / SE-2

**Lecture 10 : Layering in Practice: Separation of Concerns**

*Modules form the bridge between the logical (domain) view of the design and the development view. In this lecture we consider what modules have to 'look like' from a development point of view.*

Let's consider the appropriate module structure for supporting a multiple-model user interface, multiple-user repository.

The main responsibilities are
- display data according to the interface style (look and feel)
- obtain user requests and events according to the interface style
- control interaction (workflow) of a use case
- maintain current state of a use case ("session state")
- maintain current data of repository

We have determined to program at each Layer in a different language (C++, csh, and (PHP, C, and Java) in order to make clear which layer the material is "at". This is not uncommon and indeed often neatly defines what parts of a system lie at which layer in practice.

We look at some of the possible subsystem structures for this.

**Monolithic**
We could assign all the responsibiilities to the one Model subsystem: but this seems to lose control of separation of vertical concerns (areas of functional responsibility) and makes the one Subsystem too unstable as commands and user-interface styles come and go. [diagram] [code diagram]

The main areas of concern (subsystems / vertical slices) are
- the data respository
- each command or application
- each presentation mechanism (UIMS)

The presentation mechanisms are generally fixed by the choice of UIMSs: a Browser for HTML, Unix stdio for the CLI, and the Swing/J2SE runtime for the GUI. Presumably, then, we will create one subsystem for each command.

**Mismatch**
I mentioned the problem of architectural mismatch ("impedance mismatch") last time. Let's concentrate on that mismatch which arises between the Model layer (which is in a general purpose language, namely C++) and the Application layer (which is largely shell scripts). A shell script can receive a startup message ("exec"), with basically two arguments:
- a string of tokens (the "command line"), and
- a table of named strings (the "environment").

A script can also receive standard input (normally lines of text) and can send standard output (ditto). A script can execute jobs (including background or asynchronous jobs) and exercise some control over them. However, a script can't issue procedure or method calls.

On the other hand, we've determined that an actual variable, let's say an instance of the following class [code diagram] is the representative of the model – the job of this variable will be to know the current state of the model. The interface between this variable and the rest of the system is this class's public interface, in other words, method call and return.

Mismatch can occur at the same layer (as when reusing a module or two which were designed with different assumptions while fitting responsibilities at the same layer), but very frequently occurs between layers when using multiple languages.

Same-layer mismatch can be addressed by *adapters*, which modify the interface of a module to conform to the requirements of another module. They might *wrap*, which usually eliminates the "old" interface from the public space; or they might *mediate*, which suggests an independent module or new interface glued to the old one. More later.

Cross-layer mismatch is handled by means of a *façade*. A façade module is a module whose reponsibility is typically to present the face of a subsystem one-layer-below or sometimes one-layer-above for interaction at-this-layer. It represents the way in which clients of the subsystem can depend on it without crossing into a new layer which otherwise they are separate from.

If there is only a single session (user) active at a time, the Variable can be set up (restored from its saved persistent state) for that session alone; also it can be torn down (saved to its persistent state) when the session ends.

[collaboration diagram for **put** involving Variable and Workflow modules, single user]

**Process View**
To ensure that the single instance of the Variable is well-defined and accessible, we need to ensure that it's instantiated just once, just in one address space: this is a *process* question. (There exist techniques, such as the "singleton pattern" for ensuring this within a single program and address space; but we are discussing the multi-process architecture here.)

[process view involving Variable and Workflow processes, single user]

**Multiple Instantiation**
But when there is more than one user active at a time, this collaboration fails to meet the requirement of multi-user sharing. In the logical view we have each command's workflow coupled to the façade. This doesn't address the question of multiple instantiation.

[process view involving the Variable façade and Workflow modules, multiple user]

Some means is needed of distributing events originating with the Variable to all processes which are interested in them. There seem to be two important decisions:

1. When are application processes bound to the repository?
2. Where does the multiplexing take place?

**Binding of Applications to the Model**
Sessions (application instances) have to be created, and they have to establish communication with to the model (server) somehow.

Typically the earlier the binding, the potentially more efficient the communication can be: but the less flexibly the system can be changed or can respond to changes in load.

Binding could take place at

- compile time or before (maybe appropriate for ultra-high efficiency, e.g. in a tiny embedded kernel). In this setting there may well be coupling from the facade back to the applications
- link time (seems only appropriate for single-user case)
- system start time (a bit old-fashioned, but perhaps the server needs to allocate resources in a block from the operating system)

- Session start time (the typical command-execution pattern as in a shell or as when using the cgi protocol)
- Any run time (a more modern approach in which dynamic load balancing becomes possible)

A common logical technique for late binding is *publish-subscribe*. [collaboration diagram] In this model, a module which needs to receive events from another module indicates its need by *subscribing* to the serving module, and

[collaboration diagram changed to show subscribing and publishing events]
The issues here include

- Keep track of who's observing
- Noticing when observers go away

**Multiplexing**
In general we might multiplex events at any of the following points:

- Through the layering within the Model subsystem (utility commands, publish/subscribe)
- From façade to application (classical server)
- From application layer up to the presentation layer (servlets)

**Façade Multiplexing**
If we multiplex through the layer, there'll be multiple instances of the façade module (each of which must be a running program due to assumptions at the Application layer); each will have to have its own "copy" of some means of access to the Variable. This means is called a *proxy* because it usually represents the "real thing" and deals with access (including in this case, multiplexing). Other reasons to proxy include: crossing physical boundaries, changing security (access) protocols, and the like.

**Service Multiplexing**
If we multiplex between the façade and the application, there'll be multiple instances of the application modules (of course! we're assuming there are multiple sessions) but a single instance of the façade. In this case the façade is a *server gateway*.

**Servlet Multiplexing**
If we multiplex between the application and the presentation, there'll be many instances of the presentation modules, one for each concurrent user, each generating its own stream of user events/ But there'll be only one instance of a module which handles a given use case. (There might be multiple instances if necessary to handle load, but not in general as many as of users.) This mode is called a *servlet* because it's one of a collecction of "little" servers, one per use case, rather than one monolithic server.

**References**
Garlan, Allen, and Ockerbloom. "Architectural Mismatch or, Why it's hard to build systems out of existing parts". Proceedings of the 17th International Conference on Software Engineering (ICSE-17), April 1995 . See http://www-2.cs.cmu.edu/~able/publications/archmismatch-icse17/
Buschmann chap. 3.6