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

Lecture 5 : Software Architectures: Shared Information Systems

'Shared Information Systems' are those whose most visible structure is a central, persistent, managed stored of data, tightly coupled with a number of independent computations. The central store represents the state, at a given time, of all the system-based work which its users are doing. We look at the history and the structure of these very important systems. The designation and to a certain extent the presentation are based on Shaw and Garlan's Chapter 4, of the same title.

[Somerville sec 13.1]
A software system by its essential nature processes information (data). When it's divided up into pieces (*e.g.* subsystems), there are basically two ways to do this:

- Each subsystem maintains its own data and exchanges with other subsystems by means of a common interface. [Typical Youdon Dataflow Diagram]
- Common data is held in a central repository which all other subsystems use a common interface. [Typical Repository Diagram]

It will be often seen that the first solution is adopted first, for early or small systems, and the second solution is an evolutionary goal. The first solution requires close and careful design coordination between pieces, but less reliance is placed on the qualities of a single component. The second solution encourages the centralization of certain design decisions (e.g. data schema) but requires a database subsystem which is as reliable as the system as a whole ought to be.

In terms of *Layering*, it should be noted that the data flow approach typically involves fewer layers of the system as a whole, and (hence) requires a less layered, less sophisticated system altogether.

For example, in a simple dataflow approach, each individual program might be reading and writing ASCII text; and the master repository might be accessed through simple tape block reads and writes. In such case there may only be two layers (two "tiers") in the system: technical services and the application layer: data and control are managed by the operating system. (Today a technical infrastructure layer might interpose a standard language, probably XML, to mediate the streams, which would probably not be tapes. The architecture remains simple and somewhat inflexible.)

On the repository approach, conversely, the database core begins to constitute a *separate layer* in its own right (the technical infrasturcture) with facilities provided by a DBMS, including general data management facilities (logging, robustness, security, distribution), and data structuring and interpretation facilities also (logical and phyiscal schemas. import and export translation, physical representation control). Furthermore the DBMS will independent and more sophisticated use of the file system in lower layers (in some cases such as OS/400 the DBMS *is* the file system).

In this lecture we are otherwise presupposing the gross layered structure of the system, and focusing on slightly finer-grained detail. It's at this level of detail that most discussions of 'software architectures' are conducted.

Examples

Typical IDE

CVS

GCC

HASP ["Blackboard"]

Persistency
In some of the above examples, the data are meant to remain accessible between interactions or executions of the transactions and processes. In others, especially the compiler, the intermediate data are not retained. The former situation is called *persistent* data. Especially in object-oriented design (where data are objects whether persistent or not) a persistency facility is often and well-placed in a technical layer, typically above that of a general purpose (non-OO) database. Such a facility may provide caching, rollback and commit (at the object level), and flexibility in data "naming", as well as the mere fact of persistence.

In dataflow diagrams the double-bar notation is used to indicate a persistent repository (nonpersistent data just flow along the dataflow lines.) But the persistency is relative: compare the compiler repository to the master file respository.

The distinction between persistent and nonpersistent is less important that it might seem, because the key idea is *sharing* between multiple processes or multiple users.

Modeling Languages
The account being given here of shared-information systems is fundamentally *architectural*. That is, the details at the level of modules or objects and calsses are suppressed, and the sequence logic and communication mechanisms are little more than sketched. The UML is not very strong at this level, and design processes which use the UML (e.g. UP, RUP) seem to have less to say about design at this level. Thus Kruchten uses other notations (in the 4+1 views) for these architectural levels.

Note that these DFDs don't really specify the degree of multiprocessing or the mechanism for passing nonpersistent data. We'll discuss the various possibilities later.

Evolutionary History
This history tells the story of the technology over the last 40 years or so, and also the story of many individual systems (some of which are really that old!)

This account is based on Chapter 4.2 of Shaw and Garlan, which you should read.

In the following I'm careful not to say 'database' when I mean 'information system'. Although the terms are not used consistently in all cases, the former is more often applied to what is more fully called a database management system (DBMS), upon which information systems may be constructed.

Isolated Program
This is presumably the simplest situation of a single program which updates a file on demand. The implication is that it is not interacting with the user, but simply accepting a single command or command line, and acting accordingly.

[diagram: Isolated Program] (Process View)

Batch Sequential
When more complex processing is required, typically when many isolated programs have been created to do things to the central repository (or indeed to a collection of repositories) the maintenance problem becomes awful. The main difficulties are: all those programs handling the same data file and format, and scheduling and tracking their order and mutual dependency.

[diagram: Many Isolated Programs] (Process View)

The solution was to organize the whole into a sequence of 'batch jobs' which in the simplest form looks like this. The notion of a "transaction" replaces that of an execution of an isolated program. The system

now “knows” the set of transactions that might be performed on the repository. The user‘s input is collected for all transactions, and syntax edited.

In this presentation “on-line” means interacting with a user; its opposite is “batch” which means run in scheduled sequence. Batch jobs are run to a schedule under control of the operating system and a human operator who “starts” them. Originally the data flow was handled by scratch tapes (temporary between one job and the next) and the repositories by master tapes. Nowadays this is done with disk files or virtual files (that is, with an additional repository layer, not shown). But the batching principle is the same, namely that each batch job is run to completion before its successor(s) start.

[diagram: Batch Sequential information System] (Process View)

Error handling is always important and should be indicated in architectural plans, at least slightly. (In choosing layering plans it’s already important, because the meaning of an error or exception varies as it passes through the layers.) In these architectures, error reports are typically simply printed reports describing the failed transactions.

[diagram: Batch Update Program structure] (Development View)

Note that there are no synchronization problems here! Everything happens in a nice ordered way, two things are never happening at the same time, it can always be stopped and restarted, very relaxing...

File Types
The repository is often realized physically not as a DBMS table, but indexed files or flat files, especially in older systems.

A flat file is just a file with no indexing and probably with a fixed record structure (each line having a length and layout fixed by its content). Access to flat file data is through the familiary open/read sequentially/ close style. Sometimes open/read randomly/close is useful: random access is by “record number” which soem operating systems allow. (Not UNIX though, it only knows “byte number”.)

An indexed file is a flat file with additional (technical layer) facilities added to obtain access to its records by content. Moving towards a database but not there yet.

On-Line Transactions
The forces leading to changes in the above were and are

- the unsatisfactory delay between registering, posting, and seeing the results of a transaction
- the improvement in technical services allowing more and faster on-line processing
- the great difficulty in maintaining transactions when they are all centrally organized

Generally these forces led and lead to a more “modern” on-line system architecture.

In this architecture there are three roles, at least, which the repository is playing, each of which is normally a “database” (*i.e.* managed by a DBMS). The central repository is still the “master”. But also the DBMS is enabling synchronization of multiple transactions, and output to be scheduled for later printing. These problems occur because of the loss of strict synchroization relative to the previous architecture.

The database is also a place where exceptions can be recorded. In this architecture exceptions are being checked and corrected independently of the transactions which created them. This is a way to ensure that exceptions are handled uniformly, but it might make the user experience a little frustrating!

[diagram : Interactive Information System]

The value of sequentially and regularly-scheduled operations on an information system remains, though, so this system still has a sequential processing facilities.

Virtual Repository / MultiDatabase
The next pressures were those of corporate reorganizations, acquisitions, and mergers, which led to the need to deal with master data in multiple databases across the organization. This raises typical problems of *data migration* and *data integration* when, for example, one database represents dates as YYMMDD and another as YYDDD (year. daynumber). Or customer phone number is stored in two out of three master files and customer address in a different two out of three.

The problem cannot be solved without translation between data representations: a subject somewhat out of the scope of this already widely-scoped course. However, the solution called *multidatabase* or *federated database* is to integrate the repositories while maintaining their autonomy almost fully. A global facility is introduced to handle global queries and updates. Translation between the various representations is factored out into import/export filters and “multiple views” in the underlying DBMS technology.

[diagram]

Layered Integrated Repository

[diagram]

Web Services

[diagram]

References
Shaw and Garlan 2.6, Chapter 4.
M Shaw and P Clements. A field guide to boxology: preliminary classification of architectural styles for software systems. Proc. COMPSAC97, 21st Int'l Computer Software and Applications Conference, August 1997, pp. 6-13. Try http://spoke.compose.cs.cmu.edu/shaweb/p/pubs.htm
Somerville Ch 13 Section 1
Buschmann, "Blackboard" (pp 71-91).
Budgen sec. 6.2.3