COSC345 Week 23

Software Architecture

16 September 2014

Richard A. O'Keefe

About Real Architecture

The disappointment of Bath: architect-designed façades, builder-hacked interiors and backs.

The importance of Alexander: "pattern language" idea borrowed for coding. Major idea: rooms, buildings, streets, towns should suit the people who live and work there and should be adaptable by them.

Modern architects consider what people *do* in a house.

What is software architecture?

HIGH LEVEL design of software system.

Says HOW it all works.

Uses METAPHOR to help us understand and tell each other about the program we are working on.

Is expressed in a range of NOTATIONS.

Typically uses HIERARCHICAL decomposition.

Boxes (systems and subsystems) and arrows (control/data flow).

What do we want from an architecture?

→ Relatively simple

→ Makes change easy

→ Not tied to single platform/vendor

→ Reusable/reused

What are architectures made from?

Perry and Wolf (SIGSOFT Software Engineering Notes 17(4):40–52, 1992) meta-model:

Processing elements turn inputs into outputs

Data elements are/hold information

Connecting elements tie things together (names, paths, flows, rules)

Add: refinement relates models at different levels of detail.

Architectural styles

Shaw and Garlan *Software Architecture: Perspectives on an Emerging Discipline* characterise architectural styles:

What types of components are there?

What types of data/control connectors are there?

What constraints are there?

Examples: data flow, virtual machine, call-and-return, repository, domain-specific, process control.

Call/return architectures I

A layered architecture is a hierarchy of abstract machines each using the services of the machines below it.

Pure layering: machine *N+1* uses only machine *N*, not machine *N-1*

Example: hardware, microcode, BIOS, operating system services, programming language runtime support, `libwww`, `libxml`, XML application.

Example: operating system, X Window System "wire protocol", Xlib, Xt, Motif, your application.

Layering

Layer *N* can be tested before layer *N+1* exists. Layer *N+1* might be portable to a different implementation of layer *N*. (Tcl/Tk code portable to UNIX, Windows, MacOS as long as you don't look at lower layers.)

Sometimes modelled as DAG of modules, connections show "who calls whom".

Call/return architectures II

Traditional OOP has objects interacting via call/return.

Processing elements are methods, data elements are objects, connecting elements are inheritance, relationships, may-call.

Java/UML packages are often layered.

Process architectures I

Classical data-flow models.

UNIX pipes-and-filters models.

Systolic architectures (meshes, rings, and so on).

Distributed programming (JINI, CORBA, DCOM)

Process architectures II

Client-server architectures

Servers enrol in a registry; they provide services.

Clients find servers and send them requests.

Examples: Sun's NFS (Network File System),

Process Architectures: 3-tier

Fat client: server does data management, client does application processing and presentation.

Same machine *or* different machines.

Thin client: server does data management and application processing, client just does presentation.

Three-tier: central database server, thinnish client, in between a *middleware* layer handles authentication, load balancing *etc*.

Repository model

Shared data held in a central database.

Subsystems exchange data with the repository, little or none with each other.

Processing may be triggered by changes to data.

Example: CASE environments.

Example: some CAD environments.

See Sommerville pp 220–221 for advantages/disadvantages.

Traceability

Requirements document lists requirements. Each has a name or number.

System delegates requirements to various subsystems.

Subsystems delegate requirements to sub-subsystems. . .

Traceability matrix records which subsystems are responsible for which requirements.

Sub(sub)systems with no requirement in common shouldn't need to communicate directly.

Never forget the audit/logging requirements.

Interfaces!

Box-and-line diagrams show system structure and communication patterns.

Don't ever forget the interfaces. If subsystem A communicates control and/or data to subsystem B, then B must provide an interface and A must use it.

Do keep interfaces as small and simple as you can.

"You Ain't Gonna Need It" but "You Oughta Think About It".

Some design elements are in many places

How you make program data persistent

How you access external data stores

Security

Diagnostics, Configuration, Logging

Internationalisation and localisation.

Reading (1)

Pretty much anything in the QA76.754 area of the library.

Sommerville, *Software Engineering*, 7th edition, chapters 11 (mainly), 12 (for distributed systems), and 13 (application architecture).

Sommerville, *Software Engineering*, 6th edition, chapters 10 and 11.

Sommerville, *Software Engineering*, 5th edition, chapter 13.

Reading (2)

Pressman, *Software Engineering, a Practitioner's Approach*, section 10.6, mostly scattered elsewhere.

Hamlet and Maybee, *The Engineering of Software*, section 11.4.

Peters and Pedrycz, *Software Engineering, an Engineering Approach*, chapter 7.

Buschmann, Meunier, Rohnert, Sommerlad, & Stal, *Pattern-Oriented Software Architecture: A System of Patterns*.

Reading (3)

~ok/345/Yourdon-DFDs.pdf — 50 pages about Data Flow Diagrams

~ok/345/Yourdon_JESA.pdf — the Systems Analysis book it came from.

~ok/345/swa-sen.pdf — the Perry and Wolf article

~ok/COSC345/garlan-shaw.pdf — the Garlan and Shaw paper

The Wikipedia.