Architectual patterns: "from mud to structure"

Layers

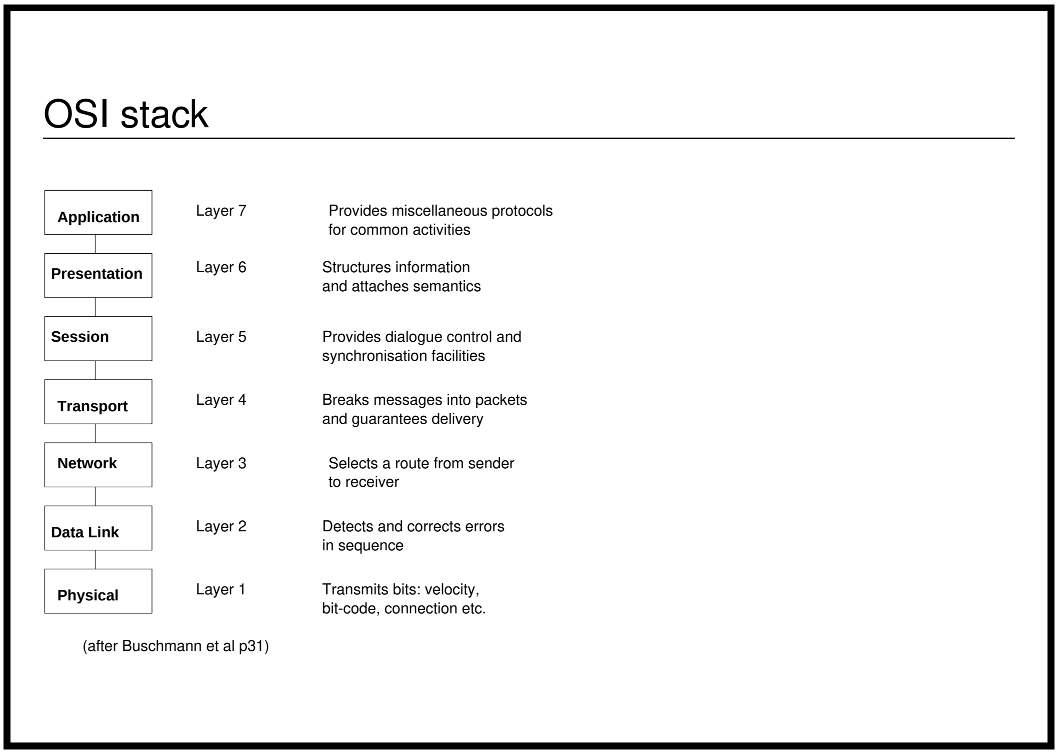
Here is an extremely abbreviated summary of Buschmann et al. p31–51.

Abstract

The Layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

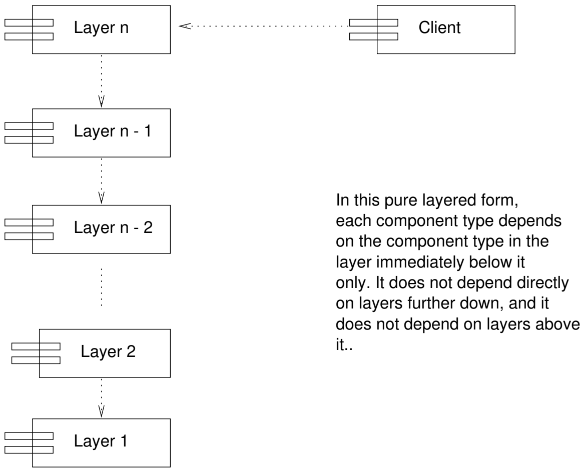
Example

Networking protocols: OSI 7-layer model is the classic example.



Slide 4

“Pure” Layers



Slide 5

Forces to balance

- Want to limit effect of late changes: isolate to small no. of components
- Interfaces should be stable (standardisable)
- Parts of the system should be exchangeable
- Lower level issues may recur: allow reuse?
- Want to group similar responsibilities to help understandability and hence maintainability
- Will have to decide granularity of components: no obvious “standard”
- Complex components need further decomposition
- Performance issues: crossing component boundaries may cost
- Team development needs clear boundaries

Slide 6

Good and bad consequences of using layers

$\surd$ Reuse of layers	X Cascades of changing behaviour
$\surd$ Support for standardisation	X Lower efficiency
$\surd$ Dependencies are kept local	X Unnecessary work
$\surd$ Exchangeability	X Difficulty of establishing granularity of layers

Slide 7

Mentions of other architectural patterns

- Pipes and Filters

  The Pipes and Filters architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent fulters. Recombining filters allows you to build families of related systems. One of the decisions to be taken is where the control over the dataflow resides: does the receiver call a method of the sender to get its data (pull model), does the sender call a method with void return type (aka a procedure) of the receiver to pass it its data (push model) or is there an active pipe? Pipes and filters is more flexible than Layers in the sense that you can recombine the filters in a different order, add and remove filters easily. Recombining layers of a layered architecture is not normally possible. However, error handling is much easier in a layered system than in a pipe and filter system.

  You might like to consider how error handling can be done in a pipes and filters system. For a start, what architectural decision is shared between Unix programs which may be combined using Unix pipes, to support error handling?

- Blackboard

  Used especially in AI systems. Useful when several components (e.g. implementing different techniques to solve the same problem or part of the same problem) must share information. Information is controlled by the Blackboard component itself; a scheduler controls the invocation of the various knowledge sources that may use and add to the information. It's often useful for the knowledge sources to run in parallel.

- Microkernel

A technique used especially in operating systems (Mach, Windows NT) in which a microkernel controls resources and communication between bigger internal servers which provide most of the functionality.

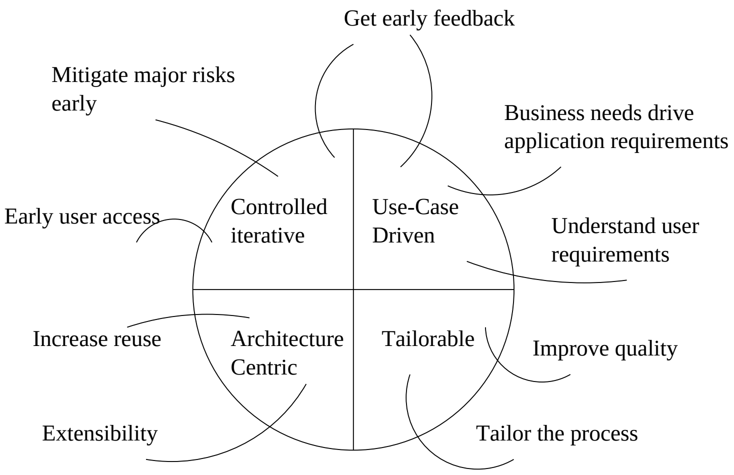
ARCHITECTURE CENTRIC
component based development

The notes in Chapter 4 of the book say what you need to know from this lecture (some of it is revision):

Book reading:
Chapter 4

but here are the slides:

Characteristics of Objectory



(adapted from Rational slide)

Slide 8

Controlled iteration

Controlled in the sense that each iteration is a mini-waterfall.

+ Mitigate risk. E.g. check user requirements, try out technology, practice new techniques in an early iteration to catch errors before main cost of project starts.

− Cost: e.g., repeated testing and documentation. A few projects are so low risk that iteration isn't cost-effective.

In practice, need for rework: essential to allow time for refactoring.

Alternatives:

- waterfall, on projects where risks are well understood, e.g. because it's very similar to a previous project (low risk).
- less controlled iteration, where correctness isn't crucial and requirements are flexible (low risk). (E.g. Linux).

Slide 9

Architecture centric COMPONENT-BASED development

(Most of what you need to know is in Chapters 4 (basics) and Chapter 18 (further consideration of components, frameworks and reuse). These notes are just a small supplement, with the slides. At this stage you should read the Panel 18.1.2 and the short section 18.3: you may prefer to read the rest of Chapter 18 too, since it will all be covered in the course in the end.

Book reading:
Chapter 4, Chapter 18

How can you get (re)use of components?

Today we'll consider some technical factors and various types of components. For example, CBD requires a clear understanding of architecture; and by some definitions parts of the architecture can *be* components.

However many of the most important factors are to do with culture, quality process etc. We'll consider those later in the course.

Slide 10

What are software components?

(We exclude for now other reusable things that in general we might call components.)

Crucial element: PLUGGABILITY

Covers both use and reuse. There is little (but perhaps not no) point in building a system from pluggable parts if you have to build all the parts first and never use them again.

- Want to have components available when we need them
- Want to be able to build big systems with few components
- Want as little as possible to do before and after the plugging: architecture-compatible components, late composition.

Slide 11

Low level interface and communication issues

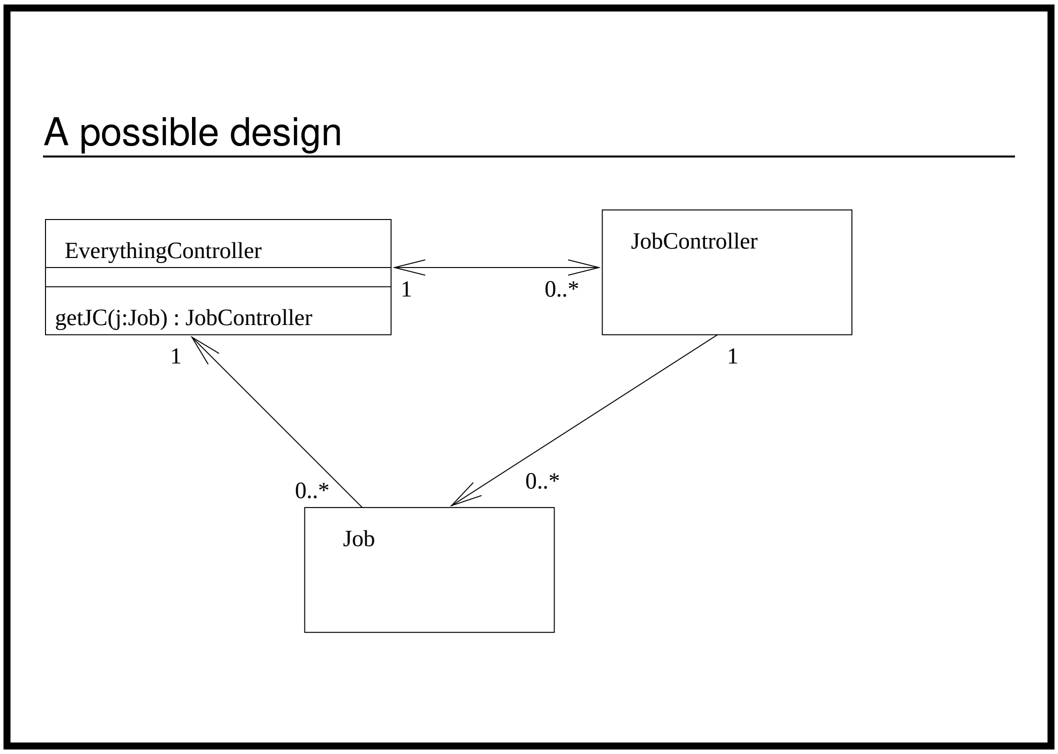
Obviously components have to

- be loosely coupled
- be highly cohesive
- have well-defined interfaces

Interfaces should limit the coupling in the system, too...

Slide 12

(If you want further explanation of the Law of Demeter it can be found in Chapter 9 of Using UML.)



Slide 13

You were asked (if there was time!) to

1. develop a sequence diagram for what happens when a Job needs to send a message to its associated JobController.
2. consider this design, bearing in mind that EverythingController has other clients apart from Jobs.
3. suggest an improvement.

The trouble is that by returning a JobController to which the requester will send messages, EverythingController exposes more than just its own interface: if the interface to JobController changes, clients of EverythingController have to be examined to see whether they need to change. So although it looks from the diagram as though we've done a good thing – avoided a two-way association between JobController and Job – in fact the two way dependency is still there, only slightly obscured, *and* we can't control other dependencies on JobController.

Law of Demeter

In response to a message m, an object O should send messages *only* to the following objects:

1. O itself
2. objects which are sent as arguments to the message m
3. objects which O creates as part of its reaction to m
4. objects which are *directly* accessible from O, that is, using values of attributes of O.

The point is, in particular, that it should *not* send messages to objects that it receives as replies to earlier messages.

Slide 14

Frameworks

or, why reuse one component when you can reuse many plus the architecture?

Two main kinds (various terminology exists, we'll use none):

- reuse by inheritance: you are provided with a bunch of classes that work together in a standard way, and you specialise the behaviour of the classes as necessary
- reuse by composition: you are provided with a bunch of classes that work together usefully but normally in a less rigid way, and you use them by composition.

Slide 15

Most real frameworks provide elements of both. GUI toolkits are the example you may have come across.

(Example in Chapter 16 of Using UML if you want one.)