COSC345
Architecture Patterns
Friday 2015.04.16
• These patterns are from chapter 2 of
• “Pattern-Oriented Software Architecture”
• by Buschmann, Meunier, Rohnert, Stal, and Sommerlad
• alias the “Gang of Five” or “Siemens” book
• There are by now five volumes, this is v1.
Layers

• “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”

• OSI network model, Brooks’ “subsumption architecture” for robots, hardware/BIOS/OS/libraries/application/scripting
Layers — Forces

- Changes should stay in one component
- Interfaces should be stable
- Parts should be interchangeable (browser)
- Similar systems with different top expected
- Work in team has to be subdivided
Layers — Tasks

• Define “abstraction”; which is a higher layer
• Decide how many layers are appropriate
• Name them and assign responsibilities
• Specify the services; layer N+1 should get all it needs from layer N, not layer N-1.
Layers — Tasks 2

- Iterate the design
- Specify the interfaces
- Structure each layer
- Decouple the layers; don’t “wire in” connections from lower to higher layers
- Figure out error handling
Relaxed Layers

- Layer N may use layers N-1, N-2, ... I directly, not just layer N-1.
- The X11 system is a good example. Or bad example. Whatever.
- Some obvious advantages; two big disadvantages are less maintainable, and harder to learn N layers than 1.
Layers — two kinds

• Classical pattern is call/return. Boxes are procedures. Arrows represent X calls Y. Abstractly boxes may be modules, abstract data types, or classes.

• You can have layers of VMs: real Pentium RISC core < x86[-64] ISA < Java VM < JavaScript VM (Rhino)...
Pipes and Filters

• “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 filters. Recombining filters allows you to build families of related systems”

• tr -cs "[:alpha:]" "\n" | tr "[:upper:]" "[:lower:]" | sort -u | comm -23 - /usr/dict/words
Pipes & filters — forces

- You are processing streams of data
- Processing is done in steps
- Data flow may split and join but no cycles
- You want to be able to compose “large” steps and small ones easily
- You may have multiple cores/cpus to use
Pipes & filters - scheme

- Boxes represent processing steps.
- Arrows represent data flowing from a source or step to a step or sink.
- Data could be characters as in UNIX or objects as in Windows PowerShell or records as in mainframe Pipe systems.
P&F Schemes 2

- Arrows are semantically bounded buffers
- Execution of processing steps may be interleaved or truly concurrent.
- Filters should normally be incremental; read a little process a little write a little. sort is exceptional.
P&F - not just for shell

- This approach requires (quasi-)concurrent processes + buffered communication channels.
- Smalltalk Processes and SharedQueues
- Erlang processes and their mailboxes
- Java Threads and ConcurrentLinkedQueues
Eager vs Lazy

• A Unix pipe-and-filter network is driven by input. Each stage pushes data to the next. In functional programming, this is eager evaluation.

• Lazy evaluation is driven by output. Nothing is read until it is needed to determine some output.
Compilers have used up to 17 different “passes” — separate programs communicating by intermediate files or intermediate pipes.

UNIX C compiler: preprocessor, front end, code generator, assembler, linker. Eager.

XML parsers are often lazy, pull driven.
P&F — Tasks

• Divide the work into processing steps
• Choose a pipe implementation~
• Design and code the filters
P&F — Tasks 2

- Consider having a configuration language for wiring up a collection of filters; this could be stylised Java code, or tables, or...

- Design the error handling. Can you avoid changing or replacing existing files until it is known to be safe?
Classical Repository

• Not a POSA-v1 pattern.
• Not to be confused with another “repository” pattern that’s basically a façade over some sort of persistence.
• This is about a system with lots of subsystems that have to communicate.
Each subsystem could manage its own data and they could send data to each other.

Or there could be a single data base, the **repository**, and each subsystem could exchange data only with the repository.
Repository—forces

- Large amounts of data
- Data are long lived
- Actions might be triggered by storing data
- You want ACID
- The subsystems should be decoupled
- *Can* the data be defined independently?
Repository—Tasks

- Enumerate the subsystems
- Design the data
- I mean REALLY put a LOT of effort into designing the shared persistent data
- Figure out who should be allowed to do what; give subsystems least possible rights
Repository—Tasks 2

• Define what counts as “well formed” data so you can detect bad data

• Define policy and procedures for correcting bad data

• Yes, that’s “error handling” by another name