

Software Architecture

Perspectives on an Emerging Discipline

CS 532 Software Design

Chapter Two Learning Objective

...to give an appreciation of *Software Architectural Styles*. Many patterns or architectural styles that exist today were developed over many years. System designers recognized the value of specific organizational principles and structures for certain classes of software. *Lets consider some and indulge in the rich space of architectural choices and understand some of the trade-offs involved.*

Frederick T Sheldon

Assistant Professor of Computer Science
University of Colorado at Colorado Springs

Chapter Two

Emerging Issues of Architectural Design

⊗ Architecture styles

⊕ SW has organizational styles

- Client-server, pipe-filter, layered
- Object-oriented and data-flow

⊕ See Fig. 2.1 for a list of common styles

-
- ⊗ A style defines a family of systems in terms of a pattern of organizational style. Effectively a vocabulary of components and connector types and a set of constraints on how to combine all of those. Also, there may exist semantic models that specify how to determine a system's overall properties from the properties of its parts

Key Architectural Issues

- ⊗ What is the design vocabulary
 - ⊕ Components and connectors
- ⊗ What are the allowable structural patterns
- ⊗ What is the underlying computational model
- ⊗ What are the:
 - ⊕ Essential Invariants of the style
 - ⊕ Common examples of its use
 - ⊕ Advantages / disadvantages
 - ⊕ Common Specializations

Pipes and Filters

See Fig. 2.2

- ⊗ Each component has a set of I/Ps and O/Ps
 - ⊕ Streams of data in and out such that output begins before the input is consumed
 - ⊕ Connectors serve as conduits for the streams
- ⊗ Invariants
 - ⊕ Filters must be independent entities
 - ⊕ Filters do not share state with other filters
 - ⊕ Filters do not know the identity of the other up/down stream filters

Pipes and Filters II

⊗ Invariants also include:

⊕ Correctness should not depend on the order that the filters perform their incremental processing

⊕ Fair scheduling may be assumed

⊕ Hugh? This means that filters may reside on a time shared system where a particular filter may get a slice of time and then need to relinquish the CPU

⊕ When each filter processes all its input data as a single entity → *Batch Sequential* (a separate style)

Pipe and Filter Examples

⊗ Unix shell scripts

⊕ E.g, the notation: `more | page filename`

⊕ Run time mechanisms are provided by Unix

⊗ Compilers are pipeline systems

⊕ Though phases are not usually incremental

⊕ Scanning, parsing, semantic analysis, and code generation

⊗ Other domains include:

⊕ DSP, parallel processing, functional programming and distributed systems

Pipe and Filter Properties

- ⊗ Overall system I/O behavior can be understood as a composition of behaviors
- ⊗ Supports reuse (Lego style plug and play)
- ⊗ Updating and adding new filters is easy
- ⊗ Natural concurrency
- ⊗ Filters may be implemented incrementally and separately (executed in parallel with other tasks)

Data Abstraction / O-O Organization

See Fig. 2.3

- ⊗ Primitive ops are encapsulated in an ADT
(which are effectively the objects)
- ⊗ Objects are managers responsible for
preserving the integrity of a resource
- ⊗ Interactions through function / procedure
invocation
- ⊗ Important facets:
 - ⊕ Preserving the integrity of its representation
 - ⊕ Representation is hidden from other objects

Object Oriented Properties

The Good News....

- ⊗ Hiding its representation form its clients:
 - ⊕ Implementation may change without affecting the clients
- ⊗ Bundling a set of accessing routines with the data they may implement:
 - ⊕ Allows designers to decompose problems
 - ⊕ Collections of interacting agents

Object Oriented Properties

The Bad News....

- ⊗ Objects must know each other to interact with one another
 - ⊕ Pipe-Filters are not constrained in this way
- ⊗ Whenever the identity of an object changes:
 - ⊕ Modify all other objects that explicitly invoke it
 - ⊕ In module-oriented langs we change the import list of every module that uses the changed module
 - ⊕ Other problems include transitive side effects:
 - A uses B and C uses B but C's affect on B looks like A's (unexpected) effect on B

Event Based / Implicit Invocation

- ⊗ Typically, component I/Fs provide a collection of procedures and functions
 - ⊕ Interaction among components \Leftrightarrow Invocation
- ⊗ Alternative integration technique:
 - ⊕ Implicit invocation
 - ⊕ Reactive integration
 - ⊕ Selective broadcast

Implicit Invocation Works

- ⊗ Components announce or broadcast one or more events
 - ⊕ Other components can register for the event
 - ⊕ The event announcement implicitly causes the invocation of procedures
- ⊗ Examples
 - ⊕ Debuggers, programming environments, DBMS(to ensure coherence/consistency)
 - ⊕ Syntax directed editors to support incremental semantic checking

Implicit Invocation

The Good News...

- ⊗ Strong support for reuse
 - ⊕ Register a new component
- ⊗ Eases system evolution
- ⊗ Interfaces are stable
 - ⊕ good for when you need to replace a component

Implicit Invocation

The Bad News....

- ⊗ Components relinquish control over the computation performed by the system
 - ⊕ When a component announces an event it cannot assume that the other components will respond
- ⊗ Sometimes data can be passed with an event
 - ⊕ However other times data exchange must rely on the use of a repository

Layered Systems

See Fig. 2.4

- ⊗ Hierarchical organization
 - ⊕ Each layer provides service to the layer above
 - ⊕ A layer serves as a client to the layer below it
 - ⊕ Works well for security purposes: special write-down program is used to breach layers.
- ⊗ Components implement a Virtual Machine at some layer in the hierarchy
- ⊗ Some layers may be partially opaque
 - ⊕ Connectors are defined by the protocols that determine how the layers actually interact

Layered Systems Examples

- ⊗ Layered communication protocols
 - ⊕ Each layer provides a substrate for communication at some level of abstraction
 - ⊕ Lower layers define lower levels of interaction right on down to the hardware
- ⊗ OS and Database systems

Layered Systems

The Good News...

- ⊗ Support for design based on increasing levels of abstraction (useful for partitioning complex design problems into incremental steps)
- ⊗ Support enhancements where changes to one layer only affect at most two other layers
- ⊗ Support Reuse through interchangeable layering
- ⊗ Standard layer interfaces may be defined
 - ⊕ E.g., ISO model and X-Window system protocols

Layered Systems

The Bad News...

- ⊗ Not always the most natural way to structure a system
- ⊗ Performance considerations
 - ⊕ Closer coupling between high level functions and their low level implementations
- ⊗ Difficulty in finding the right levels of abstraction (e.g., mapping existing protocols to into the ISO framework)

Repositories

See Fig. 2.5

- ⊗ Two distinct types of components:
 - ⊕ Central data structure (I.e., the state)
 - ⊕ Collection of independent components that operate on the central data store.
- ⊗ Interaction among these entities can vary significantly. Two major subcategories:
 - ⊕ Traditional database
 - ⊕ Blackboard (works like a trigger)

Repositories → Blackboard

⊗ Knowledge sources

- ⊕ Independent parcels of application-dependent knowledge

⊗ Blackboard data structure

- ⊕ Problem solving state data (changes to the blackboard incrementally lead to the solution of the problem)

⊗ Control

- ⊕ Driven by the state of the blackboard

Repositories

There Is Only Good News...

- ⊗ Naturally they are not a panacea
- ⊗ Many examples abound
 - ⊕ Speech and pattern recognition (shared access to data with loosely coupled agents)
 - ⊕ Batch-sequential systems with global databases
 - ⊕ Programming environments organized as a collection of tools
 - ⊕ Most compilers operate on a base of shared information (symbol table, syntax trees, etc)

Interpreters

See Fig. 2.6

- ⊗ Virtual machine is produced in the software
 - ⊕ Includes the *pseudo-program* being interpreted and the *interpretation engine*
- ⊗ Pseudo-program
 - ⊕ Program itself and the interpreters analog of its execution state (activation record)
- ⊗ Interpretation engine
 - ⊕ Definition of the interpreter and current state of its execution

Interpreter → Four Components

- ⊗ *Interpretation engine* to do the work
- ⊗ A memory that contains the *pseudo-code* to be interpreted
- ⊗ A representation of the *control state* of the interpretation engine
- ⊗ Representation of the current state of the program being *simulated*

Interpreter Domain

- ⊗ Used to build virtual machines that close the gap between
 - ⊕ Computing engine expected by the semantics of the program
 - ⊕ Computing engine available in hardware
- ⊗ Programming language are sometimes referred to as providing:
 - ⊕ A virtual *Pascal* machine
 - ⊕ A virtual *Java* machine

Process Control

See Figs. 2.7 - 2.11

⊗ Based on process control loops

⊕ Characterized both by the kinds of components involved and the special relations that must hold among them.

⊕ See Figure 2.7 for definitions:

- Process variables and controlled variable
- Input variable and manipulated variable
- Set point

⊕ System types include:

- Open-loop / closed-loop / feed-back and feed-forward control

Process Control Mechanisms

- ⊗ Converting input materials to products
 - ⊕ Specific properties of the output require operations being performed on the inputs and intermediate products
 - ⊕ *Controlled variables of the process*: variables that measure properties of the output materials
 - ⊕ Manipulated variables are associated with things that can be changed by the control system to regulate the process

Control System Purpose

(Hint → Process Control)

- ⊗ Maintain specified properties of the outputs at (sufficiently near) the given reference values (set points)
- ⊗ Open-loop
 - ⊕ Process runs without surveillance (materials are pure and the operations are repeatable)
- ⊗ Closed-loop
 - ⊕ Properties are monitored (e.g., flow rates, temp., pressure)
 - ⊕ Control comes in by changing the settings of the apparatus (valves, heaters, and chillers)

Process Control Closed-loop

Two General Forms

⊗ Feedback control (figure 2.10)

⊕ As discussed

⊗ Feedforward control (figure 2.11)

⊕ Anticipates future effects on the controlled variable by measuring variables whose values may be more timely

⊕ Useful when lags in the process delay the effect of control changes

Process Control ← Software Paradigm

- ⊗ The normal model corresponds to an *open-loop* system
- ⊗ If external disturbances affect the software system, control paradigm should be considered:
 - ⊕ Computation elements
 - Process definition and control algorithm(s)
 - ⊕ Data elements
 - Process variables, set points and sensors
 - ⊕ Control loop paradigm

Other Familiar Architectures

- ⊗ Distributed processes characterized by:
 - ⊕ Topological features such as ring or star
 - ⊕ Others in terms of IPC protocols
 - ⊕ Client-Server (who-knows-who / RPC)
- ⊗ Main-program / subroutine organizations
 - ⊕ Systems mirror of the programming language
 - ⊕ Main program is driver

Other Familiar Architectures II

- ⊗ Domain-specific software architectures
 - ⊕ Reference architectures tailored to a family of applications
 - Avionics
 - Command and control
 - Vehicle management systems
 - ⊕ Synthesis from the architectural description
 - In many cases the Architecture is sufficiently constrained that an executable system can be generated automatically (or semi-automatically)

Other Familiar Architectures III

⊗ State transition systems

- ⊕ Common for many reactive systems

- ⊕ Defined in terms of named transitions that move a system from state to state

Heterogeneous Architectures

- ⊗ Most systems are *not pure*
- ⊗ *Combining styles is possible in several ways*
 - ⊕ Internal / external embedding of styles
 - ⊕ Switchboard
 - Permit a single component to use a mixture of architectural connectors (e.g., Unix uses repository and pipe-filter mixtures)
 - Active database repository + implicit invocation
 - ◆ External components register for interest in portions of the database. The database automatically invokes the appropriate tools on the basis of this association

Heterogeneous Architectures II

⊗ Combining styles

- ⊕ Completely elaborate one level of an architectural description in a completely different style