Topic 04: Software Architecture: Intro and Styles
Zhen Ming (Jack) Jiang
Acknowledgement

Some of the slides are from Richard C. Holt, Spiros Mancoridis, and Emad Shihab
References


- The Architecture of Open Source Applications
The Evolution of Programming Abstractions

The first modern programmable computers (1950s) were largely hardwired. The first software was written in machine language.

Next major breakthrough: assembly languages
- Symbolic assemblers
- Macro processors

1960s: High-level languages (3GLs)
- Mostly independent of machine and problem domain
  - Level is “generic problem-solving”
- FORTRAN, COBOL
- Algol, Pascal, Modula
- C, PL/1
- Simula, Smalltalk; C++, Java
Abstraction from Developers’ Perspective

- Typed variables and user-defined types [late 1960s]
- Modules [early 1970s]
  - Create explicit interfaces, enforce information hiding
- ADTs and object-oriented computing [mid 1970s]
  - Programming entities as mathematically precise constructs
  - Abstract commonalities to one place
- Object-oriented design patterns, refactoring [1990s]
  - OO is powerful and complex
    - What constitutes a “good” OO design (small to medium-sized programs)?
    - What re-usable “tricks” can help to solve recurring problems?
  - At the level of data structures, algorithms and a few co-operating classes
- Software architecture [1990s, but really since 1960s]
  - Designing large systems is about understanding broad tasks, defining system-wide structures, interfaces and protocols, understanding how non-functional requirements impact on the system
  - At the level of the handful of “big boxes” that comprise the major components of your system, plus their interdependencies
Software Architecture

- As the size and complexity of software systems increases, the design problem goes beyond algorithms and data structures.
- Designing and specifying the overall system structure (Software Architecture) emerges as a new kind of problem.
- Reference Architecture
  - General architecture for an application domain. E.g., common structure for compilers or for operating systems
- Product Line Architecture (PLA)
  - Architecture for a line of similar software products. E.g., software structure for a family of computer games
Software Architecture Issues

- Organization and global control structure,
- Protocols of communication, synchronization, and data access,
- Assignment of functionality to design elements,
- Physical distribution,
- Selection among design alternatives.
State of Practice

- Currently there is no well-defined terminology or notation to characterize architectural structures.
- However, good software engineers make common use of architectural principles when designing complex software.
- These principles represent rules of thumb or idiomatic patterns that have emerged informally over time. Others are more carefully documented as industry standards.
“Camelot is based on the client-server model and uses remote procedure calls both locally and remotely to provide communication among applications and servers.”

“We have chosen a distributed, object-oriented approach to managing information.”
“Abstraction layering and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices. The architecture encourages a client-server model for the structuring of applications.”
“The easiest way to make a canonical sequential compiler into a concurrent compiler is to **pipeline** the execution of the compiler phases over a number of processors. A more effective way is to split the source code into many segments, which are concurrently processed through the various phases of compilation (by multiple compiler processes) before a final, merging pass recombines the object code into a single program.”
Some Standard Software Architectures

- **ISO/OSI Reference Model** is a layered network architecture.

- **X Window System** is a distributed windowed user interface architecture based on event triggering and callbacks.

- **NIST/ECMA Reference Model** is a generic software engineering environment architecture based on layered communication substrates.
The “Toaster” Model
Intuition About Architecture

- It is interesting that we have so few named software architectures. This is not because there are so few architectures, but so many.

- We next look at several architectural disciplines in order to develop an intuition about software architecture. Specifically, we look at:
  - Hardware Architecture
  - Network Architecture
  - Building Architecture
Hardware Architecture

- **RISC** machines emphasize the instruction set as an important feature.
- **Pipelined and multi-processor** machines emphasize the configuration of architectural pieces of the hardware.
Differences & Similarities
Between SW & HW Architectures

■ Differences:
  – relatively (to software) small number of design elements.
  – scale is achieved by replication of design elements.

■ Similarities:
  – we often configure software architectures in ways analogous to hardware architectures. (e.g., we create multi-process software and use pipelined processing).
Network Architecture

- Networked architectures are achieved by abstracting the design elements of a network into nodes and connections.

- Topology is the most emphasized aspect:
  - Star networks
  - Ring networks
  - Manhattan Street networks

- Unlike software architectures, in network architectures only few topologies are of interest.
Building Architecture

- **Multiple Views:** skeleton frames, detailed views of electrical wiring, etc.
- **Architectural Styles:** Classical, Romanesque, and so on.
- **Materials:** One does not build a skyscraper using wooden posts and beams.
What are Architectural Styles

- An **Architectural Style** defines a family of systems in terms of a pattern of structural organization. It determines:
  - the **vocabulary** of components and connectors that can be used in instances of that style
  - a set of **constraints** on how they can be combined.
Why Architectural Styles

- Makes for an easy way to communicate among stakeholders
- Documentation of early design decisions
- Allow for the reuse and transfer of qualities to similar systems
Architectural styles are named so we can easily discuss & contrast them, not so they can be carried on banners or elevated to purity.

Roy T. Fielding @fielding

Event Based Integration vs REST is a great discussion. So is EBI + REST and EBI REST, especially on mobile platforms with built-in EBI.

Roy T. Fielding @fielding

Just don’t assume EBI will work on platforms that don’t have it built-in; it requires careful balance of social, economic, and tech factors.

Greg L. Turnquist @gregturn

@fielding its vocabulary, not dogma.

Matthew Cannon @matt_p_cannon

@fielding @keithb_b Exactly this! Even the old GoF patterns had expressive metaphorical names like Bridge and Visitor. There was a reason.
Determining an Architectural Style

- We can understand what a style is by answering the following questions:
  - What is the **structural pattern**? (*i.e.*, components, connectors, constraints)
  - What is the underlying **computational model**?
  - What are the essential **invariants** of the style?
  - What are some common **examples** of its use?
  - What are the **advantages** and **disadvantages** of using that style?
  - What are some of the common **specializations** of that style?
Describing an Architectural Style

- The architecture of a specific system is a collection of:
  - computational components
  - description of the interactions between these components (connectors)

- Software architectures are represented as graphs where **nodes** represent components:
  - procedures
  - modules
  - processes
  - tools
  - databases

- and **edges** represent connectors:
  - procedure calls
  - event broadcasts
  - database queries
  - pipes
Architecture Styles Covered

- Pipe and filter
- Repository
- Implicit invocation
- Layered
- Client-server
- Process-control
Pipe and Filter Style
Pipe and Filter Architectural Style

- Suitable for applications that require a defined series of independent computations to be performed on ordered data.

- A component reads streams of data on its inputs and produces streams of data on its outputs.
Pipe and Filter Architectural Style (Cont’d)

- **Components**: called filters, apply local transformations to their input streams and often do their computing incrementally so that output begins before all input is consumed.

- **Connectors**: called pipes, serve as conduits for the streams, transmitting outputs of one filter to inputs of another.
Pipe and Filter Architectural Style
Pipe and Filter Invariants

- Filters **do not share state** with other filters.
- Filters **do not know the identity** of their upstream or downstream filters.
- The **correctness** of the output of a pipe and filter network should not depend on the order in which their filters perform their incremental processing.
Pipe and Filter Specializations

- **Pipelines**: Restricts topologies to linear sequences of filters.

- **Batch Sequential**: A degenerate case of a pipeline architecture where each filter processes all of its input data before producing any output.
Pipe and Filter Examples

- **Unix Shell Scripts:** Provides a notation for connecting Unix processes via pipes.
  - `cat file | grep Eroll | wc -l`

- **Traditional Compilers:** Compilation phases are pipelined, though the phases are not always incremental. The phases in the pipeline include:
  - lexical analysis + parsing + semantic analysis + code generation
Pipe and Filter Style
Advantages & Disadvantage

Advantages
- **Easy to understand** the overall input/output behavior of a system as a simple composition of the behaviors of the individual filters.
- They **support reuse**, since any two filters can be hooked together, provided they agree on the data that is being transmitted between them.
- Systems can be **easily maintained and enhanced**, since new filters can be added to existing systems and old filters can be replaced by improved ones.
- They permit certain kinds of **specialized analysis**, such as throughput and deadlock analysis.
- The naturally **support concurrent execution**.

Disadvantages
- Not good for handling **interactive systems**, because of their transformational character.
- Excessive parsing and unparsing leads to **loss of performance** and **increased complexity** in writing the filters themselves.
Summary of Pipe-and-Filter Style

- Independent components connected by pipes that route data streams between filters

- Advantages
  - Easy to understand
  - Easy to maintain and enhance

- Disadvantages
  - Poor performance
  - Increased complexity
Repository Style
Repository Style

- Suitable for applications in which the central issue is establishing, augmenting, and maintaining a complex central body of information.
- Typically the information must be manipulated in a variety of ways. Often long-term persistence is required.
Components:
- A central data structure representing the current state of the system.
- A collection of independent components that operate on the central data structure.

Connectors:
- Typically procedure calls or direct memory accesses.
Repository Style (Cont’d)
Repository Style Specializations

- Changes to the data structure trigger computations.
- Data structure in memory (persistent option).
- Data structure on disk.
- Concurrent computations and data accesses.
Repository Style Examples

- Information Systems
- Central Code Repository Systems
- Programming Environments
- Graphical Editors
- Database Management Systems
- Games (World of Warcraft)
Repository Style Advantages

- **Efficient** way to store large amounts of data.
- **Sharing** model is published as the repository schema.
- **Centralized management:**
  - backup
  - security
  - concurrency control
Repository Style Disadvantages

- Must agree on a data model a priori.
- Difficult to distribute data.
- Data evolution is expensive.
Summary of Repository Style

- Independent components (programs) access and communicate exclusively through global repository

Advantages
- Efficient storage of data
- Easily manageable
- Can solve complex problems

Disadvantages
- Evolving data is expensive
- Cannot handle high volume or complex logic
Case Study
- The Architecture of a Compiler
Case Study:
Architecture of a Compiler

■ The architecture of a system can change in response to improvements in technology.

■ This can be seen in the way we think about compilers.
Early Compiler Architectures

- In the 1970s, compilation was regarded as a sequential (batch sequential or pipeline) process:
Early Compiler Architectures

- Most compilers create a separate symbol table during lexical analysis and used or updated it during subsequent passes.
Later, in the mid 1980s, increasing attention turned to the intermediate representation of the program during compilation.
Hybrid Compiler Architectures

- The new view accommodates various tools (e.g., syntax-directed editors) that operate on the internal representation rather than the textual form of a program.

- Architectural shift to a repository style, with elements of the pipeline style, since the order of execution of the processes is still predetermined.
Hybrid Compiler Architectures

Lex Syn Sem Opt Cgen

Attributed Parse Tree
Symbol Table

Edit Flow
Implicit Invocation Style
Implicit Invocation Style

Publish-Subscribe

Event Based

Taylor et al. 2010
Implicit Invocation Variants

- **Publish-Subscribe**
  - Subscribers register to receive specific messages
  - Publishers maintain a subscription list and broadcast messages to subscribers

- **Event-Based**
  - ICs asynchronously emit and receive "events" communicated over event bus
Implicit Invocation Style

■ Components
  – Publishers, subscribers
  – Event generators and consumers

■ Connectors
  – (PS) Procedure calls
  – Event bus
Implicit Invocation Style Topology

- Subscribers connect to publishers directly (or through network)

- Components communicate with the event bus, not directly to each other
Implicit Invocation Advantages

- (PS) Efficient dissemination of one-way information
- Provides strong support for **reuse**
  - Any component can be added, by registering/subscribing for events
- **Eases system evolution**
  - Components may be replaced without affecting other components in the system
Implicit Invocation
Disadvantages

■ (PS) Need special protocols when number of subscribers is very large

■ When a component announces an event:
  – it has no idea what other components will respond to it,
  – it cannot rely on the order in which the responses are invoked
  – it cannot know when responses are finished
Implicit Invocation Examples

- Used in **programming environments** to integrate tools:
  - Debugger stops at a breakpoint and makes that announcement
  - Editor scrolls to the appropriate source line and highlights it

- Twitter, Google+
QA evaluation for Implicit Invocation

- **Performance**
  - (PS) Can deliver 1000s of msgs
  - Event bus: how does it compare to Repository?

- **Availability**
  - Publisher needs to be replicated

- **Scalability**
  - Can support 1000s of users, growth in data size

- **Modifiability**
  - Easily add more subscribers, change in message format affects many subscribers
Layered Style
Layered Style

- Architecture is separated into ordered layers
  - A program in one layer may obtain services from a layer below it
Layered Style Specializations

- Often exceptions are made to permit non-adjacent layers to communicate directly.
  - This is usually done for efficiency reasons.
Layered Style Examples

Layered Communication Protocols:
- Each layer provides a substrate for communication at some level of abstraction.
- Lower levels define lower levels of interaction, the lowest level being hardware connections (physical layer).

Operating Systems
- Unix
## Unix Layered Architecture

<table>
<thead>
<tr>
<th>System Call Interface to Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
</tr>
<tr>
<td>Plain File</td>
</tr>
<tr>
<td>File System</td>
</tr>
<tr>
<td>Protocols</td>
</tr>
<tr>
<td>Cooked Block Interface</td>
</tr>
<tr>
<td>Raw Block Interface</td>
</tr>
<tr>
<td>Raw TTY Interface</td>
</tr>
<tr>
<td>Cooked TTY</td>
</tr>
<tr>
<td>Line Disc.</td>
</tr>
<tr>
<td>Network Interface</td>
</tr>
<tr>
<td>Block Device Driver</td>
</tr>
<tr>
<td>Character Device Driver</td>
</tr>
<tr>
<td>Hardware</td>
</tr>
</tbody>
</table>
Layered Style Advantages

- **Design**: based on increasing levels of abstraction.

- **Enhancement**: since changes to the function of one layer affects at most two other layers.

- **Reuse**: since different implementations (with identical interfaces) of the same layer can be used interchangeably.
Layered Style Disadvantages

- Not all systems are easily structured in a layered fashion.
- Performance requirements may force the coupling of high-level functions to their lower-level implementations.
Client-Server Style
Client-Server Style

- Suitable for applications that involve distributed data and processing across a range of components.
Client-Server Style

- **Components:**
  - **Servers:** Stand-alone components that provide specific services such as printing, data management, etc.
  - **Clients:** Components that call on the services provided by servers.

- **Connector:** The network, which allows clients to access remote servers.
Client-Server Style

- Client 1
- Client 2
- ... (Ellipsis)
- Client N

- Network
- File Server
- Database Server
- Object Server
Client-Server Style Examples

- **File Servers:**
  - Primitive form of data service.
  - Useful for sharing files across a network.
  - The client passes request for files over the network to the file server.
Here are some examples of **Database Servers**:

- More efficient use of distributing power than file servers.
- Client passes SQL requests as messages to the DB server; results are returned over the network to the client.
- Query processing done by the server.
- No need for large data transfers.
- Transaction DB servers also available.
Client-Server Style Advantages

- Distribution of data is straightforward,
- Transparency of location,
- Mix and match heterogeneous platforms,
- Easy to add new servers or upgrade existing servers.
Client-Server Style
Disadvantages

- No central register of names and services -- it may be hard to find out what services are available
Process-Control Style
Process-Control Style

- Suitable for applications whose purpose is to maintain specified properties of the outputs of the process at (sufficiently near) given reference values.
Process-Control Style

■ Components:
  – **Process Definition** includes mechanisms for manipulating some process variables.
  – **Control Algorithm** for deciding how to manipulate process variables.

■ Connectors: are the data flow relations for:
  – **Process Variables**:
    • *Controlled variable* whose value the system is intended to control.
    • *Input variable* that measures an input to the process.
    • *Manipulated variable* whose value can be changed by the controller.
  – **Set Point** is the desired value for a controlled variable.
  – **Sensors** to obtain values of process variables pertinent to control.
Feed-Back Control System

- The controlled variable is measured and the result is used to manipulate one or more of the process variables.
Open-Loop Control System

- Information about process variables is not used to adjust the system.
Process Control Examples

- Real-Time System Software to Control:
  - Automobile Anti-Lock Brakes
  - Nuclear Power Plants
  - Automobile Cruise-Control
MAPE-K loop
Exercise
Automated Stock Trading System

- A customer approaches YOURSTRULYTradingSolutions with the need for an architecture design for an automated stock trading system. The system needs to take in a list of stocks, related to specific sectors and buy/sell these stocks based on some predefined algorithms.

- The system needs to perform well (place many orders) and scale to support many investors.

- Your task is to propose an architecture for the system, accompanied by an informal evaluation of the advantages and disadvantages of the proposed architecture.
Automated Stock Trading System

- **Input**
  - List of sectors/stocks to trade
  - List of investors and their daily budget

- **Output**
  - Notification to buyers/sellers
    - List of stocks bought/sold
    - Cost, if bought; Gain/Loss, if sold
  - Withdraw/deposit money based on orders
Automated Stock Trading System

Assumptions

■ Historical data of all stocks will be provided (very large)
  – To perform historical analysis on the stocks/sectors

■ Buy/Sell algorithms will be provided by financial engineers
  – To know which stocks to buy/sell

■ Investor personal data will be provided
  – To withdraw and deposit money