EECS4314 week 3

Jerry Wu

2024-01-24

Contents

Lecture 1

Software Architecture

1.1 Evolution of programming abstractions

- First computers used punchcards and were hardwired
- First software was written in assembly
- In the 1960s, high level programming languages like Fortran, Cobol, Pascal, C, etc. were introduced to make code more readable and easier to use/understand
- ADTs and OOP in the mid 1970s
- Design patterns first introduced in 1990s, allows object reuse

1.1.1 Why software architecture?

- As software size and complexity increases, the design problem goes beyond data structures and algorithms
- Designing and specifying the overall system structure emerges as a new kind of problem
- Recall reference architecture and product line architecture

1.1.2 Software architecture issues

- Organization and global control structure
- Protocols of communication, synchronization, and data access
- Assignment of functionality to design elements

- Physical distribution of product
- Selection among design alternatives

1.1.3 State of practice

- Currently, there is no well defined terminology or notation to characterize architectural structures and systems
- Good software engineers will make common use of architectural principles when designing complex software systems
- These are simply principles or idiomatic patterns that have emerged over time

1.1.4 Descriptions of software architectures

- **Example 1**
  - Camelot is based on the **client server model** and uses remote procedure calls both locally and remotely to provide communication among appplications and servers
  - We have chosen a distributed **object oriented** approach to managing information
- **Example 2**
  - **Abstraction layering** (layered architecture) and system decomposition provide the appearance of system uniformity to clients, yet allow HeliX to accomidate a diversity of autonomous devices (self managing systems like server management, not self driving cars)
  - The architecture encourages a **client server model** for structuring of applications
- **Example 3**
  - The easiest way to make a canonical **sequential** compiler into a **concurrent** compiler is to **pipeline** the execution of the compilar 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

1.1.5 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. Linux can run without a GUI, but it can have a windowed GUI thanks to this.
- **NIST/ECMA reference model** is a generic software engineering environment architecture based on layered communication substrates

1.2 Intuition about architecture

We need to look at several architectural disciplines to develop an intuition about software architecture. Specifically:

- Hardware architecture
- Network architecture
- Building architecture

1.2.1 Hardware architecture

- RISC machines emphasize the instruction set as an important feature
- Pipelined and multiprocessor machines emphasize the configuration of architectural pieces of the hardware

1.2.2 Differences between hardware and software architecture

- **Differences**
  - Relatively small (compared to software) number of design elements
  - Scale is achieved by replication of design elements
- **Similarities**
  - We often configure software architectures in an analogous way to hardware. E.g. we create multi process software by using pipelining and multithreading

1.2.3 Software vs 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 topology
  - Ring Topology
  - Manhattan street networks
- Unlike software architectures, network architectures only have a few topologies of interest

1.2.4 Software vs building architeture

- **Multiple views**
  - Skeleton frames, detailed views of electric wiring, etc.
  - SW: class UML, use cases, etc
- **Architectural styles**
  - Classical, roman, etc.
  - SW: pipelines, layers, etc.
- **Materials**
  - Do not build a skyscraper using wooden posts and beams
  - SW: algorithms (e.g. which sorting algorithm to use)

1.2.5 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 beused in instances of that style
  - A set of constraints on how said components can be combined together into a system

1.2.6 Why bother with 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

1.3 Types of architectural styles

1.3.1 Disclaimer

There is no architectural style that is a silver bullet. Every style has their advantages and disadvantages.

1.3.2 Pipe and filter

- 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
- **VERY COMMON FOR COMPILERS**
- **Components**: called **filters**
  - apply local transformations to their input streams
  - they 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 the inputs of another
- **Invariants of pipes and filters**
  - Filters do not share states with other filters
  - Filters do not know the identity of their upstream or downstream filters
  - The correctness of output of a pipe and filter network should not depend on the order in which their filters perform their incremental processing

- **Specializations**
  - Pipelines: Restricts topologies to **linear sequences** of filters
  - Batch sequential: A degenerate case of pipeline architecture where **each filter processes all of its input data before producing any output**

1.3.3 Examples of pipe and filter

- UNIX shell scripts: Provides a notation for connecting UNIX processes via pipes. E.g. `cat file | grep Eroll | wc -l`
- Traditional compilers: Compilation phases are pipelined though the phases are not always incremental. The phases in the pipeline inclide:
  - lexical analysis
  - parsing
  - semantic analysis
  - code generation

1.3.4 Advantages and disadvantages of P&F

- **Advantages**
  - **Easy to understand** the overall input/output behavior of a system; composition of the behaviors of the individual filters
  - They **support reuse** since any two filters can be hooked together; common data formats between them
  - Systems can be **easily maintained and enhanced**; new filters can be added and old filters can be replaced
  - They permit certain kinds of **specialized analysis** like throughput and deadlock analysis
  - They naturally support **concurrent execution**
- **Disadvantages**
  - Not good for handling **interactive systems** due to their transformational character
  - Excessive parsing and unparsing leads to **loss of performance** and **increased complexity**

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. **Persistent data/storage is required**
- In a nutshell, pretty much anything which has a **database, or is a management system**
- **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