

Software Architecture

Perspectives on an Emerging Discipline

CS 531 SW Requirements Specification and Analysis

Chapter One Learning Objective

- . . . to give an appreciation of *Software Architecture* as an emerging and important facet of the upstream portion of the *Software Life Cycle*. As a phase that comes after requirement elicitation/specification and before *Software Design*, it's an important tool/discipline useful to the software engineering practitioner.
-

Frederick T Sheldon
Assistant Professor of Computer Science
University of Colorado at Colorado Springs

Chapter One

Emerging Issues of Architectural Design

What is Software Architecture?

Structural and organizational issues about systems

- Global control, communication protocols, synchronization and data access
- Allocation of resources (e.g., function design elements)
- Design element composition (info hiding, coupling, cohesion)
- Physical distribution
- Scaling and performance
- Dimensions of evolutions
- Selection among design alternatives

Example Architectures

Client - Server model

Remote Procedure Call (RPC) structuring

Abstraction layering

Distributed Object Oriented approach

Pipeline - Filter framework

Architecture Patterns

Collection of idioms, patterns, and styles of software system organization that serves as a shared, semantically rich vocabulary

Example Pipelined Architecture:

Streamed transformation

Function behavior can be derived compositionally from the behavior of constituent filters

Frameworks for Understanding

Software architecture structures serve as frameworks for understanding the big picture (broader issues):

- System level concerns

- Global flow rates, patterns of communication

- Executive control structure, scalability

- System evolution

Properties can be fleshed out

Architecture of Software Systems

Defines the system in terms of computational components and interactions among the components

Components are...

- Clients / Servers
- Databases
- Filters
- Layers in a hierarchy of elements/components

Software Design Levels

(See Fig. 1.1)

Architecture

Overall association of system capability with components

Code

Algorithms, data structures, language primitives, etc...

Executables

Memory maps, stacks, register allocations, ISAs

Problem is, SW is understood at the level of

Intuition

Anecdote

Folklore

One Possible Solution

Improve the precision of understanding at the SW Architecture level

Programs, modules, systems

- Rich collection of *interchange representations* and protocols to connect components and system patterns to guide the compositions

An Engineering Discipline for Software

What is engineering?

Creating cost effective solutions . . .

. . . to practical problems . . .

. . . by applying scientific knowledge . . .

. . . building things . . .

. . . in the service of mankind.

Engineering . . .

Relies on *codifying scientific knowledge* about a technological problem domain

Provides answers for common questions that occur in *practice*

Engineering *shares prior solutions* rather than relying on virtuoso problem solving

Enabling ordinary practitioners to create sophisticated systems that work

SW success (diligence+hard work) and failures (poor understanding+mismatch of problem with solution)

Current SW Practice

Knowledge about techniques that work is not shared

Comp. Sci. has contributed some relevant theory but practice proceeds largely independently!

Therefore,

There are fundamental problems with the use of the term *software engineer*

Practitioners recognize the need for ways to share experience with good designs

Routine and Innovative Design

Routine design involves solving familiar problems

Reusing portions of prior solutions.

Innovative design involves finding novel solutions to unfamiliar problems.

Software in most application domains is treated more often as original than routine...

Certainly more so than would be necessary if we captured and organized what we already know!

Model for Evolution of an Engineering Discipline

(Fig. 1.2)

Engineering emerges from the commercial exploitation that supplants craft

Exploiting technology depends on . . .

Scientific engineering

Management

Marshaling of resources

Engineering must return workable solutions!

Engineering generates good problems for science and science, after finding good problems in the needs of practice, returns workable solutions.

Maturity of Supporting Science

Research on ADTs:

Specifications (abstract models and algebraic axioms)

Software Structure (bundling representations with algorithms)

Language issues (protecting integrity of information not in specifications)

Integrity constraints (invariants of data structures)

Rules for composition (declarations)

The whole field of computing is only 40 yrs0 old... many theories are emerging in the research pipeline

Interaction Between Science and Engineering

(Figure 1.3 Codification Cycle for Science and Engineering)

Models and theories

Improved practice

New problems

- Ad hoc solutions
- Novel solutions

Folklore

Codification

Models and theories

Evolution of Software Engineering

(Figure 1.4 Evolution of Software Engineering)

Where does current SE practice lie on the path to engineering?

In some cases it's a craft

Yet in others it's a commercial practice

And, in isolated examples, one could argue that professional engineering is taking place!

Codification Through Abstract Mechanisms

Conversion from an intuition (i.e., get the data structure right [ADT]) to a theory involve understanding the following:

The software structure (a representation packaged with its primitive operators)

Specifications (mathematically expressed as abstract models or algebraic axioms)

Language issues (modules, scope, user-defined types)

Integrity of the result (invariants of data structures and protection from other manipulation)

Rules for combining types (declarations)

Information hiding (protection of properties not explicitly included in specifications)

Just as good programmers recognized useful data structures in the late 60's, good SW system designers now recognize useful system organizations. One of these is based on the theory of abstract types. But, this is not the only way to organize a SW system.

Building Composable Systems

Flexible - high level connections between existing systems in ways not foreseen

Similar to what developers of “open” software products have designed

Interchange representations

- PICT, RTF, SYLK and SGML, HTML
- To allow distinct products to interact by data interchange.
- CORBA supports dynamic sharing

Large Systems Decompositional Mechanisms

Tractability is a problem with large systems

Breaking a system into pieces makes it possible to reason about the overall properties by understanding the properties of each part

MILs and IDLs have traditionally helped:

- Computational units with well-defined interfaces
- Compositional mechanism for gluing the pieces together

MIL/IDL: What's the nature of the glue?

The Purpose of the “glue” is to resolve

Definition / use relationships

Indicate for each use of a facility where its corresponding definition is provided

Maps well to current programming languages.

Good for the compiler

Supports automated checks (type checking)

Formal reasoning (pre- and post- conditions)

Drawbacks of MIL/IDLs

Implementation Interaction relationships!

Status of Software Architecture

The Bad News...

Useful architecture paradigms are typically only understood in an idiomatic way and applied in an ad hoc fashion

SW Architects have been unable to

- exploit commonalties in system architectures

- Make principled choices among design alternatives

- Specialize general paradigms to specific domains

- Teach their craft to others

Status of Software Architecture

The Good News...

The issues and problems are being addressed in such areas as...

Module interface languages (MIL)

Domain specific architectures

Software reuse

Codification of organizational patterns for SW

Architecture description languages

Formal underpinnings for architectural design

Architectural design environments

A Sound Basis for SW Architecture

Benefits for both development and maintenance:

Recognize common paradigms new systems can be built on variations of old systems

The right architecture is crucial to development success

Detailed understanding of the SW architecture enables the engineer to make principled choices

Fluency in software architecture notations and paradigms SW Engineer to communicate the new system design

Impediment to Reuse

Differences in component packaging

Differences in packaging are recognized only informally!!

There exists no formal (or informal) guidance that show how and when to use such packaging

It is often unclear whether components with compatible functionality will actually be able to interact properly

Some Open Problems

Choosing the appropriate architecture for a given problem or domain

Rules of style - dictate how to package components.

- E.g., as procedures, objects, or filters and often cannot be interchanged across styles!

Interfaces make incompatible assumptions

- E.g., in UNIX sort is available as a filter and a procedure.

Open Architectures

Some architectures are carefully documented and widely disseminated

ISO's interconnection reference model

NIST/ECMA Reference model (PCTE)

- Generic SEE framework

X-Windows (distributed Window I/F architecture)

- Based on event triggering and callbacks

To Summarize

Primary considerations

Understanding architectural abstractions

Localizing and codifying the ways components interact

Distinguish the various ways architectural principles can be applied to

- Software system design
- Analysis

Use what good engineers have always found useful in practice