


# Origins of Software Architectures


1



## The Origins

- ❑ Software Architecture emerged as a discipline in 1992
  - Paper by Perry and Wolf (ironically not well received initially)
- ❑ Software Engineers have always employed software architectures
  - Most often without realizing it!
  - Architecture meant different things to different people
  - Not standardized
  - Not broadly understandable

2




## The Origins (cont.)

- ❑ Address issues identified by researchers and practitioners
  - Essential software engineering difficulties
  - Unique characteristics of programming-in-the-large
  - Need for software reuse (principles)
  - Computer architecture ≠ software architecture
  - Stop re-inventing the field
- ❑ Architectures incorporate solutions
  - Module interconnection languages
  - Megaprogramming (cutting edge 20 years ago!)
  - Formal specification methods & languages
  - Transformational programming

We talk about these today

3



## Pewter Bullets


- ❑ Ada, C++, Java & other high-level languages
- ❑ Object-Oriented Programming, Design, Analysis
- ❑ Artificial Intelligence
- ❑ Automatic, Graphical Programming
- ❑ Program Verification
- ❑ Environments & tools
- ❑ Workstations instead of mainframes

SW gets complex faster than we are able to catch up

now we are pretty good at solving the problems of 20 years ago...

- ❑ Others? Did the web solve our problems?


4



## Promising Attacks On Complexity (In 1987)

- Buy vs. Build
  - If possible, better to buy than build
- Requirements refinement & rapid prototyping
  - Hardest part is deciding what to build (or buy?)
  - Must show product to customer to get complete specification
  - Need for iterative feedback


5



## Promising Attacks On Complexity (In 1987)

- Incremental/Evolutionary/Spiral Development
  - Grow systems, don't build them
  - Good for morale
  - Easy backtracking
  - Early prototypes
- Great designers
  - Good design can be taught
  - Great design cannot
  - Nurture great designers


6



## Programming In The Large (“PITL”)

- DeRemer and Kron in 1976 Paper
- Two distinct software development activities
  - PITS = programming; PITL = software engineering
- Two Phases:
  - Structuring large collections of modules to build systems
  - Developing individual modules
    - Programming languages typically support this task only
    - E.g., how do you know whether Class A calls Class B? Structural information is dispersed throughout the system
    - E.g., how do you know what a class requires?
    - E.g., how do you know what *not* to do with a class?
    - E.g., how do you know what the expect result will be? Method `int plus(int a, int b)`


7



## Programming In The Large (cont.)

- Structural information is dispersed throughout system
  - Modules
  - Procedure calls
  - Linker instructions
  - Documentation
- Difficult to understand how to compose/extend system
  - Boundaries
  - Interfaces
  - Provided services
  - Assumptions
- 30 years old but there is still no programming language/environment


8



## What To Do?

- Use different languages for different activities
  - Allow software to be developed
    - From heterogeneous parts?
    - By different people?
  - Interpersonal dynamics become factor in software development


9



## What To Do? (cont.)

- PITL techniques should focus on at least these concerns
  - Project management
    - To structure interaction among team members (not everybody can work on everything)
  - Software design
    - To establish effective overall system structure (inter-module dependencies)
  - Communication
    - To enable interaction among team members
  - Documentation
    - To enable communication, understanding, evolution, & maintenance


10



## A Possible Solution

- Use high-level language to specify system structure
  - Concise, precise, & checkable
    - but also understandable, easy to learn, use, not verbose
  - Act as linker of separately compiled/produced modules
- Show visibility of information & services
  - Dataflow and controlflow
- Specify connectivity
  - Allow access to shared information & resources
  - Hide/restrict access to information resources
  - Define direct communication
  - Define no communication
- Minimize module interdependencies (Why?)
  - And enforce it! (compile-time, link-time, run-time)

11



## Module Interconnection Languages (MIL's)

- Languages for programming-in-the-large
- Provide formal description of software system's global structure
  - Defines system modules
  - Specify how modules fit together
    - Interconnections may be data or control
  - Concise, precise, & verifiable
  - More work, Less freedom... but worth it

12

**Center For Software Engineering**

## Module Interconnection Languages (cont.)

- ❑ Assumptions of MIL's
  - System has been analyzed, designed, & evaluated
  - Individual modules have been implemented
- ❑ Main MIL concepts
  - Separate language to describe overall system structure
  - Static type checking at inter-module level
  - Control of different system versions & families

13

**Center For Software Engineering**

## MIL Terminology

- ❑ Resource
  - Smallest unit addressed
    - e.g., Function to open a file
- ❑ Module
  - Related resources doing a single task
    - e.g., Functions & data items for file access
- ❑ System
  - Hierarchically organized modules
    - e.g., a compiler
- ❑ Family
  - Set of systems that perform essentially same task
  - Variations
    - Number & type of resources
    - Implementation languages

14

**Center For Software Engineering**

## Example MIL Specification

```

module ABC
  provides a, b, c
  requires x, y
  consists-of
    function XA, module YBC
end ABC

function XA
  must-provide a
  requires x
  has-access-to module Z
  real x, integer a
end XA

module YBC
  must-provide b, c
  requires a, y
  real y, integer a, b, c
end YBC
end ABC
    
```


15

**Center For Software Engineering**

## What MIL's Don't Do

- ❑ Functional system specification
  - Only show static system structure
  - Do not specify nature of dynamic change
- ❑ Type specification
  - All types checked syntactically by a MIL
  - Type specification validity ensured elsewhere
- ❑ Module Implementation
- ❑ Loading
  - Assume existence of loaders or other similar facilities


16



## Software Reuse

- ❑ Software systems are rarely entirely new
  - Usually “variations on a theme”
- ❑ Building “one-of-a-kind” systems is expensive
  - True in any branch of engineering
  - Particularly in software engineering
- ❑ Plethora of existing solutions to “new” problems
  - (Core) Assets (Legacy Systems)
  - Off-the-shelf (OTS) systems (web server, browser)
    - May only provide partial solutions
- ❑ Line of code ceases to be fundamental software building block
  - Module/component becomes unit of development, functionality, evolution, maintenance, & reuse (need for glue-code)
  - Analogy to civil engineering (brick house versus skyscraper)


17



## Benefits Of Reuse

- ❑ Reduced development time/cost
  - Potential for “plug-and-play”
- ❑ Improved qualities
  - Reliability
    - Thorough testing
    - Multiple uses
  - Portability?
  - Interoperability?
  - Reconfigurability?
- ❑ User programmable
  - Via component composition
- ❑ But there are economical and technical drawbacks...


18



## Economic Context Of Software Reuse

- ❑ Designing for reuse requires higher up-front investment
  - Development costs increase between 10% and 50%
  - Additional costs for maintaining libraries of reusable assets
  - Maintenance of reusable components
  - *(most functionality of MS Word remains unused by the average user)*
- ❑ Costs are recouped when component is reused
  - Cost savings factors range from ~2 to ~20
  - Requires long-term vision (
  - Buy-in from management is necessary
- ❑ OTS reuse entails certain risks
  - Lack of trust
  - Uncertain reliability of reused software
  - Inadequate understanding of reused software
  - Possibility of cost and budget overruns
  - *(open-source, garage-developed 3D imaging software)*


19



## Technical Difficulties Case Study

- ❑ Architectural Mismatch paper by Garlan et al. at CMU 10 years ago
- ❑ Want to build a Software Architecture Toolkit
  - GUI, some middleware, and a database
  - What to use open-source, C++, off-the-shelf software
  - Found packages that fit the specs well
  - Integration was estimated to be 2 grad students for 1 year
- ❑ Result
  - 5 grad students over 2 years
  - System was difficult to use, very slow
  - The project failed
  - Developers were smart people!
- ❑ Contribution:
  - Architectural Mismatch Analysis


20



## Technical Difficulties With Reuse

- OTS systems may not contain clearly identifiable components
- OTS component granularity may be too coarse or too fine
- OTS components do not provide exact set of functions required
- Specialization & integration of OTS components is unpredictably complex
- Costs associated with reusing component may be higher than developing it anew
  - locating/selecting (use keywords on Google?)
  - Understanding (two components, each assuming to be the central control)
  - retrieving
  - evaluating/specializing
  - Integrating
  - versioning


21



## Software Reuse Truisms [Krueger 1992]

- For reuse technique to be effective
  - Must reduce cognitive distance between
    - Initial concept of a system
    - Final executable implementation
  - Must be easier to reuse artifact than to build from scratch
- To select artifact for reuse,
  - Must be able to find it faster than can build it
  - Must know what it does
    - If you need to look at its source code to understand it...


22



## Megaprogramming

- Conceptual software development framework
- Combines ideas:
  - Component-based development
  - Software reuse
  - Product-lines
  - Domain-specific focus
- Shifts focus from lines of code to components & their composition into systems
- Searches for commonalities across families of systems
- Moves in direction of conventionalized system structures & standards
- Alleviates problems of reuse in general


23



## Economics of Megaprogramming

- Recognize & integrate canonical software reuse roles into development organizations
  - Product line manager
  - Component manager
  - Component producer
  - Component broker
  - Component user
- Change organizational incentive structure to support reuse
  - How do you want to reward reusability?
- Educate for reuse & megaprogramming
- Effective large-scale reuse requires functioning component marketplace
  - Benefits of reuse would vastly overshadow risks
  - Standards are needed


24



## Approaches To Software Reuse (1)

- High-level languages
  - Reuse of assembly code instruction patterns
- Design &/or code scavenging
  - Requires tremendous investment time/effort
  - Benefits are unpredictable


25



## Approaches To Software Reuse (2)

- Source code components
  - Components modeled & developed specifically for reuse
  - Successful in small, well understood domains
  - General-purpose component libraries tend to be unwieldy
- Software schemas
  - Reuse of abstract algorithms & data structures
    - rather than source code
  - Formally specified at level of abstraction above code
  - May be too complex to locate, understand, & use


26



## Approaches To Software Reuse (3)

- Application generators
  - Akin to programming language compilers
    - But in narrow domains
  - Applied on very high-level, special-purpose abstractions
  - Not applicable to a broad range of applications

27



## Approaches To Software Reuse (4)

- Very high-level languages & transformational systems
  - Use “executable specification languages”
  - Typically mathematical abstractions
    - e.g., Set theory
  - More generally applicable, but not as powerful as generators
  - (Human-guided) transformation from specification to implementation
- Software architectures

28

Center  
For  
Software  
Engineering

---

## Formal Methods

- ❑ Formal method =  
Specification Language + Formal Reasoning
- ❑ Body of techniques supported by
  - Precise mathematics (not natural language)
  - Powerful reasoning tools
- ❑ Rigorous mechanisms for system
  - Modeling
  - Synthesis
  - Analysis

Some think of formal  
methods as architectures

29

Center  
For  
Software  
Engineering

---

## Formal Methods in Use

- ❑ Types of formalisms
  - Predicate logic (for all, exists)
  - Discrete mathematics (discrete objects)
  - Finite state machines
- ❑ Applicability in software development
  - System models
  - Requirements specifications
  - Constraints
  - Designs
  - (semi)automated implementation
- ❑ Desirable effects
  - Highly reliable, secure, safe systems
  - Clarify customer's requirements
  - Reveal ambiguity, inconsistency, incompleteness
  - More efficient production

30

Center  
For  
Software  
Engineering

---

## Formal Specification Language Categories

- ❑ Axiomatic
  - Operations defined by logical assertions
  - e.g., Anna
- ❑ State transition
  - Operations defined in terms of computational states & transitions (ordering)
  - e.g., StateCharts
- ❑ Abstract model
  - Operations defined in terms of well-defined mathematical model (function mapping)
  - e.g., Z
- ❑ Algebraic
  - Operations defined by equivalence relations
  - e.g., Larch

$0 < x < 10$

Town  
residences : P ADDR  
residents : ADDR ⇔ P NAME  
residences = dom residents

$f(g(x)) = f(x)$

31


Center  
For  
Software  
Engineering

---

## Transformational Systems In A Nutshell

- ❑ Recognition
  - Programming is a difficult task; characterized by problem of mastering complexity
- ❑ Premises
  - Correct programs can be constructed if task is divided into sufficiently small & formally justified steps
  - Many steps can be automated
- ❑ Conclusion
  - If automated development steps are performed by machine, programmer is free to **focus on creative aspects**
- ❑ Idealistic Goal: Requirements to code
- ❑ Realistic Goals
  - General support for program modification
  - Program synthesis from formal specification
  - Program adaptation to different environments
  - Verification of program correctness


32

 Center For Software Engineering

## Transformation Systems Terminology

- ❑ Transformation
  - Relation between two programs
- ❑ Transformation rule
  - Mapping from one program to another that constitutes correct transformation
    - Expression-substitution rules (replacement)
    - Refinement rules
    - Constant discovery and propagation
    - Dead-variable elimination
    - Loop unraveling and fusion
    - Recursion elimination
    - ...


33

 Center For Software Engineering

## Transformation Systems Terminology (cont.)

- ❑ Transformational programming
  - Program construction by successive application of transformation rules
    - Guarantees that final version of program satisfies initial formal specification
  - Ordering is relevant


34

 Center For Software Engineering

## Types Of Transformational Systems

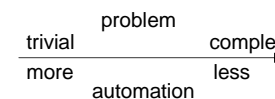
- ❑ Focus
  - Synthesis – different input and output language
  - Optimization – same input and output language
- ❑ Input language
  - Specification language
  - Programming languages
- ❑ Level of automation
  - Fully automatic
  - Semi-automatic
  - User-driven
- ❑ Transformation mechanism
  - Catalog approach – knowledge-based systems
  - Generative set approach – elementary transformations used in constructing new rules (machine code to assembly code to ... x-generation)

35


 Center For Software Engineering

## Do Transformational Systems Work?

- ❑ Fully automated transformational systems are infeasible
- ❑ Dubious usefulness & usability
  - Extremely difficult to use
  - Typically used on “toy” problems
- ❑ Require extensive expertise
  - In given formal method
  - In tools that compose system
- ❑ Generated systems are inefficient
  - Code size is larger
  - Execution speed is slower
- ❑ Generated systems are difficult to debug
  - Race conditions



36

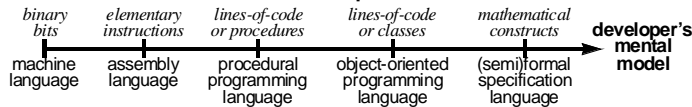


Center  
for  
Software  
Engineering

---

## Summary

- **Software is inherently complex**
- **Some of the complexity is controllable**
  - Elevate the level of the abstractions provided to developers
  - **Strive to match the developers' mental models**



- **Specific techniques are provided for this**
  - Notations to describe software systems
  - Techniques and tools to construct them from reusable, coarse-grain building blocks
  - Focus on overall system structure

STILL A LONG WAY TO GO!

37