The Origins

- For many years, software engineers have been employing software architectures without knowing it!
- Origins of *explicit* architectures lie in issues encountered and identified by researchers and practitioners
  - essential software engineering difficulties
  - unique characteristics of programming-in-the-large
  - need for software reuse
- Origins of *explicit* architectures also lie in solutions developed to deal with those issues
  - module interconnection languages
  - megaprogramming
  - formal specification methods and languages
  - transformational programming

Software Engineering Difficulties

- Software engineers deal with a unique set of problems
  - young field with tremendous expectations
  - building of vastly complex, but intangible systems
  - software is not useful on its own
  - it must conform to changes in other engineering areas
- Some problems can be eliminated
  - these are Brooks’ “accidental difficulties”
- Other problems can be lessened, but not eliminated
  - these are Brooks’ “essential difficulties”
Accidental Difficulties

- Solutions exist, they just may not have been discovered
- Past productivity increases are a result of overcoming accidental difficulties
  - Inadequate programming constructs and abstractions
    - remedied by high-level programming languages
    - increase in productivity by a factor of five
    - this complexity was never inherent in a program at all
  - Lack of immediacy in viewing one’s decisions in action
    - remedied by time-sharing
    - turnaround time eventually decreases beyond the limit of human perception
  - Difficulty of using heterogeneous programs together
    - addressed by integrated software environments
    - support a task that was conceptually always possible

Essential Difficulties

- Only partial solutions exist for them, if any
  - Complexity
    - no two software parts are alike
    - complexity grows non-linearly with size
  - Conformity
    - software is always required to conform
    - often the “last kid on the block”
  - Changeability
    - software is viewed as infinitely malleable
    - change originates with new applications, users, laws, machines
  - Invisibility
    - the reality of software is not embedded in space
    - software is not representable as a familiar geometric entity
Pewter Bullets

- Ada and other high-level languages
- Object-oriented programming
- Artificial intelligence
- Automatic programming
- Graphical programming
- Program verification
- Environments and tools
- Workstations

Promising Attacks on Complexity (in 1987)

- Buy vs. build
  - if possible, it is always better to buy than build
- Requirements refinement and rapid prototyping
  - most difficult is deciding what to build
  - must show the product to the customer to get a complete specification
  - need for iterative feedback.
- Incremental development
  - grow systems, don’t build them
  - good for morale
  - easy backtracking
  - early prototypes
- Great designers
  - good design can be taught
  - great design cannot
  - grow great designers
Programming in the Large (PITL)

- Two distinct software development activities
  1) structuring large collections of modules to build systems
  2) developing individual modules
→ Programming languages typically support only task #2
- Structural information is dispersed throughout the system
  - modules
  - procedure calls
  - linker instructions
  - documentation
- Difficult to understand how to compose/extend a system
  - boundaries
  - interfaces
  - provided services
  - assumptions

What to Do

- Use different languages for the different activities
  - allow software to be developed from heterogeneous parts
  - interpersonal dynamics become a factor in software development
- PITL techniques should focus on at least these concerns
  - project management
    → to structure interaction among team members
  - software design
    → to establish an effective overall system structure
  - communication
    → to enable interaction among team members
  - documentation
    → to enable communication, system understanding, and evolution/maintenance
A Possible Solution

- A high-level language to specify system structure
- Concise, precise, and checkable
  - but also understandable
- Show visibility of information
- Specify connectivity
- Specify disconnectivity
  - information hiding
  - restricted access to resources
- Act as a linker of separately compiled/produced modules
  → Minimize module interdependencies

Software Reuse

- Software systems are rarely entirely new
  - they are “variations on a theme”
- Building “one-of-a-kind” systems is too expensive
  - true in any branch of engineering
  - particularly the case in software engineering
- A plethora of existing solutions to “new” problems
  - off-the-shelf (OTS) systems
  - may only provide partial solutions
- Line of code ceases to be the fundamental software building block
  - module/component becomes the unit of development, functionality, evolution/maintenance, and reuse
  - analogy to civil engineering
Benefits of Reuse

- Reduced development time
  - potential for “plug-and-play”
- Improved reliability
  - thorough testing
  - multiple uses
- Improved quality
  - portability
  - interoperability
  - rapid reconfigurability
- User programmability
  - via component composability

Economic Context of Software Reuse

- Designing for reuse requires a higher up-front investment
  - development costs increase between 10% and 50%
  - additional costs in maintaining libraries of reusable assets
  - maintenance of reusable components
- These costs are recouped when a component is reused
  - cost savings factors range from ~2 to ~20
  - requires a long-term vision
    - buy-in from the management is necessary
- OTS reuse entails certain risks
  - lack of trust
  - uncertain reliability of reused software
  - inadequate understanding of the reused software
    - possibility of cost and budget overruns
Technical Difficulties in Attempting Reuse

- OTS systems may not contain clearly identifiable components
- OTS component granularity may be too coarse or too fine
- OTS components do not provide the exact set of functions required
- Specialization and integration of OTS components is unpredictably complex
- The costs associated with reusing a component may be higher than developing it anew
  - locating/selecting
  - understanding
  - retrieving
  - evaluating/specializing
  - integrating

Software Reuse Truisms [Krueger 1992]

- For a reuse technique to be effective, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation
- For a reuse technique to be effective, it must be easier to reuse an artifact than to build it from scratch
- To select an artifact for reuse, one must know what it does
- To reuse a software artifact effectively, one must be able to find it faster than (s)he could build it
Approaches to Software Reuse (1)

- High-level languages
  - reuse of assembly code instruction patterns
- Design and/or code scavenging
  - requires tremendous time/effort investment
  - benefits are unpredictable
- Source code components
  - components modeled and developed specifically for reuse
  - successful in small, well understood domains
  - general-purpose component libraries tend to be unwieldy
- Software schemas
  - reuse of abstract algorithms and data structures, rather than source code
  - formally specified at a level of abstraction above code
  - may be too complex to locate, understand, and use

Approaches to Software Reuse (2)

- 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
- Very high-level languages and 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
Module Interconnection Languages (MILs)

- Languages for programming in the large
- Provide a formal description of the global structure of a software system
  - state what the system modules are
  - specify how the modules fit together in a system
  - interconnections may be data or control
  - concise, precise, and verifiable

- Assumptions of MILs
  - the system has been analyzed, evaluated, and designed
  - individual modules have been implemented

- Main MIL concepts
  - separate language to describe overall system structure
  - static type checking at the inter-module level
  - control of different system versions and families

MIL Terminology

- Resource
  - smallest unit addressed
  - e.g., function to open a file

- Module
  - related resources doing a single task
  - e.g., functions and data items for file access

- System
  - hierarchically organized modules
  - e.g., a compiler

- Family
  - set of systems that perform essentially the same task
  - variations
    - number and type of resources
    - implementation languages
Example MIL Specification

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

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

What MILs Don’t Do

- Functional system specification
  - only show static system structure
  - do not specify the nature of its resources or 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
- Embedded linking instructions
  - assume OS services or a separate command language
Megaprogramming

- A conceptual software development framework
- Unites a number of ideas
  - component-based development
  - software reuse
  - product-lines
  - domain-specific focus
- Shifts focus from lines of code to components and their composition into systems
- Searches for commonalities across families of systems
- Moves in the direction of conventionalized system structures and standards
- Alleviates the problems of reuse in general

Economics of Megaprogramming

- Recognize and integrate into organizations the canonical software reuse roles
  - product line manager
  - component producer
  - component broker
  - component user
- Change organizational incentive structure to support reuse
- Educate for reuse and megaprogramming
- Effective large-scale reuse requires a functioning component marketplace
  - benefits of reuse would vastly overshadow the risks
  - standards are needed
Formal Methods

- **Formal Method** = Specification Language + Formal Reasoning
- Body of techniques supported by
  - precise mathematics
  - powerful reasoning tools
- Rigorous mechanisms for system
  - modeling
  - synthesis
  - analysis

Formal Methods in Use

- Types of formalisms
  - predicate logic
  - discrete mathematics
  - finite state machines
- Applicability in software development
  - system models
  - constraints
  - requirements specifications
  - designs
  - (semi)automated implementation
- Desirable effects
  - highly reliable, secure, safe systems
  - clarify customer’s requirements
  - reveal ambiguity, inconsistency, incompleteness
  - more efficient production
Formal Specification Language Categories

- **Axiomatic**
  - operations defined by logical assertions
  - e.g., Anna

- **State transition**
  - operations defined in terms of computational states and transitions
  - e.g., State Charts

- **Abstract model**
  - operations defined in terms of a well-defined mathematical model
  - e.g., Z

- **Algebraic**
  - operations defined by equivalence relations
  - e.g., Larch

Transformational Systems in a Nutshell

- **The recognition**
  - programming is a difficult task characterized by the problem of mastering complexity

- **The premises**
  - correct programs can be constructed if the task is divided into sufficiently small and formally justified steps
  - many of those steps are automatable

- **The conclusion**
  - if the automatable development steps are performed by a machine, the programmer is free to focus on creative aspects

- **The goals**
  - general support for program modification
  - program synthesis from a formal specification
  - program adaptation to different environments
  - verification of program correctness
Terminology

- Transformation — a relation between two programs
- Transformation rule — a mapping from one program to another that constitutes a correct transformation
  - expression-substitution rules
  - refinement rules
  - constant propagation
  - dead-variable elimination
  - loop unraveling and fusion
  - recursion elimination
- Transformational programming — program construction by successive application of transformation rules
  - guarantees that the final version of the program satisfies the initial formal specification

Types of Transformational Systems

- Focus
  - optimization — same input and output language
  - synthesis — different 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
Do Transformational Systems Work?

- Fully automated transformational systems are infeasible
- Dubious usefulness and usability
  - extremely difficult to use
  - typically used on “toy” problems
- Require extensive expertise
  - in a given formal method
  - in tools that compose the system
- Generated systems are inefficient
  - code size is larger
  - execution speed is slower
- Generated systems are difficult to debug

Summary

- Software is inherently complex
- Some of the complexity is controllable
  - elevate the level of abstractions provided to developers
  - strive to match 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!