Review — 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

Review — Essential Difficulties

- At best, only partial solutions exist
  - complexity
  - conformity
  - changeability
  - invisibility
- Hey, this bullet is not silver!
  - high-level languages
  - OO programming
  - AI
  - automatic programming
  - graphical programming
  - program verification
  - environments and tools
  - workstations
- Some promising attacks on complexity
  - buy vs. build
  - requirements refinement and rapid prototyping
  - incremental development
  - grow great designers

Review — Programming in the Large (PITL)

- Structuring large collections of modules to build systems
- Treat structural information as a first class artifact
- PITL foci
  - project management
  - software design
  - communication
  - documentation

→ Solution

- MIL — a high-level language to specify system structure
- allow software to be developed heterogeneously
- interpersonal dynamics become critical

- Problem
  - a very limited role
  - becomes a factor too late in development
    - → software engineering ≠ software programming

Review — Software Reuse

- Software components as units of development, functionality, evolution/maintenance, and reuse
  - reduced development time and cost
  - improved reliability and quality
  - potential for user programmability
- Economic issues
  - designing for reuse requires a higher up-front investment
  - requires a long-term vision and buy-in from the management
  - $$ benefits of reuse must outweigh the $$ risks
-

Review — Megaprogramming

- A s/w development framework that unites ideas of
  - software reuse
  - product lines
  - component-based development
  - domain-specific approaches
- Shifts focus to components and their compositions
- Aims for conventionalized structures and standards
- Economic issues
  - recognize the canonical reuse roles
  - change organizational incentive structure
  - educate for reuse and megaprogramming
  - build a component marketplace
- Great idea but still needs an accompanying methodology
  - not there yet

Review — Formal Methods

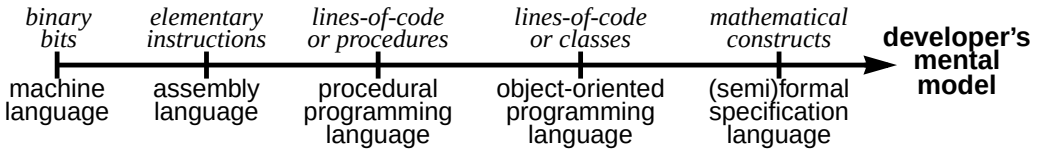
- Body of software specification techniques supported by precise mathematics and reasoning tools
- Applicability in software development
  - system models
  - constraints
  - designs
  - requirements specifications
  - automated implementation
- Desirable effects
  - reliable, secure, safe systems
  - clarify customer's requirements
  - reveal ambiguity, inconsistency, incompleteness
  - more efficient production
- Problems
  - difficult to understand
  - typically impractical for large problems

Review — Transformational Systems

- Goals
  - general support for program modification
  - program synthesis from a formal specification
  - program adaptation to different environments
  - verification of program correctness
- Transformational programming guarantees that the final program satisfies the initial formal specification
- Several problems
  - fully automated transformational systems are infeasible
  - extremely difficult to use
  - typically used on “toy” problems
  - require extensive expertise
  - generated systems are inefficient
  - generated systems are difficult to debug

Where Now?

- Control inherent software complexity
  - elevate abstraction levels
  - match developers’ mental models



→ Explicitly address a system’s conceptual *architecture*
  - modifying a completed building is difficult
  - modifying its blueprint is easy in comparison

→ *Software architecture is a software system’s blueprint*
  - addresses complexity
  - increases reuse and component marketplace potential
  - subsumes formal methods

Focus and Scope of Software Architectures

- Two primary foci
  - system structure
  - correspondence between requirements and implementation
  - → components + rules of composition + rules of behavior
- A framework for understanding system-level concerns
  - global rates of flow
  - communication patterns
  - execution control structure
  - scalability
  - paths of system evolution
  - capacity
  - throughput
  - consistency
  - component compatibility

Definitions of Software Architecture

- Perry and Wolf
  - Software Architecture = { Elements, Form, Rationale }

- Shaw and Garlan
  - Software architecture [is a level of design that] involves
    - → the description of elements from which systems are built,
    - → interactions among those elements,
    - → patterns that guide their composition,
    - → and constraints on these patterns.
- Kruchten
  - Software architecture deals with the design and implementation of the high-level structure of software.
  - Architecture deals with abstraction, decomposition, composition, style, and aesthetics.

Why Architecture?

- A key to reducing development costs
- A shift in developer focus
  - component-based development philosophy
  - explicit system structure
- Separation of concerns
- A natural evolution of design abstractions
  - structure and interaction details overshadow the choice of algorithms and data structures in large/complex systems
- Benefits of explicit architectures
  - a framework for satisfying requirements
  - technical basis for design
  - managerial basis for cost estimation & process management
  - effective basis for reuse
  - basis for consistency and dependency analysis

Key Architectural Concepts

- Three canonical building blocks
  - components
  - connectors
  - configurations
- Ideally, building blocks are defined independently
  - supports reuse in different contexts
  - supports interconnections unforeseen by original developers
  - → difficult in practice

Components

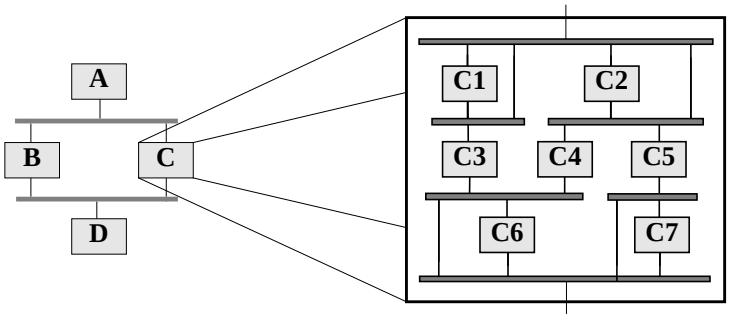
- A ***component*** is a unit of computation or a data store
  - Perry & Wolf's processing and data elements
- Components are loci of computation and state
  - clients
  - servers
  - databases
  - filters
  - layers
  - ADTs
- A component may be simple or composite
  - composite components describe a system

Connectors

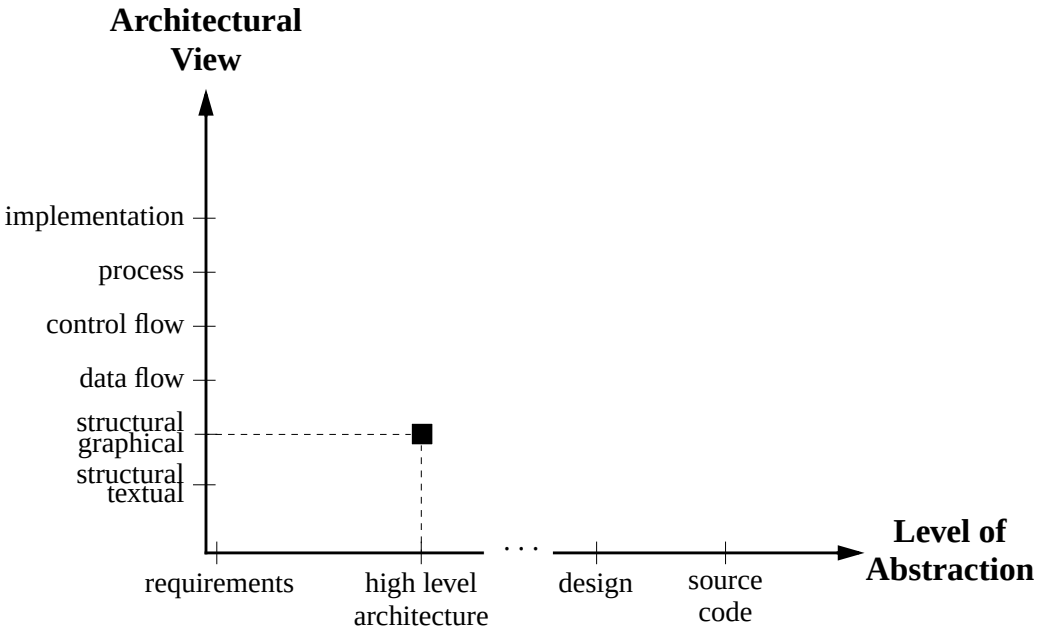
- A ***connector*** is an architectural element that models
  - interactions among components
  - rules that govern those interactions
- Simple interactions
  - procedure calls
  - shared variable access
- Complex and semantically rich interactions
  - client-server protocols
  - database access protocols
  - asynchronous event multicast
  - piped data streams

Configurations/Topologies

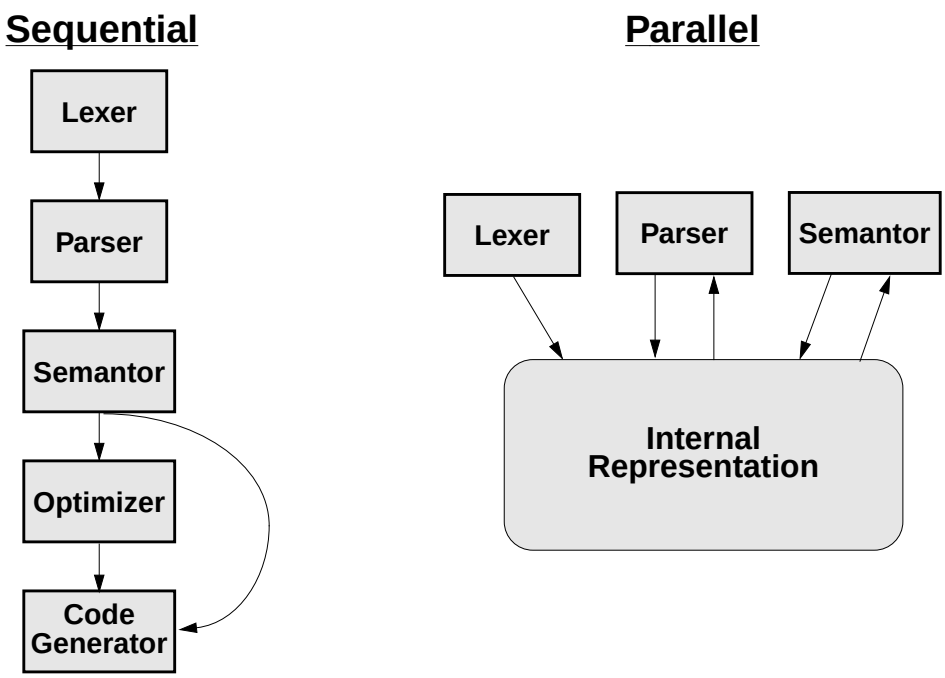
- An ***architectural configuration*** or ***topology*** is a connected graph of components and connectors which describes architectural structure.
  - proper connectivity
  - concurrent and distributed properties
  - adherence to design heuristics and style rules
- Composite components are configurations



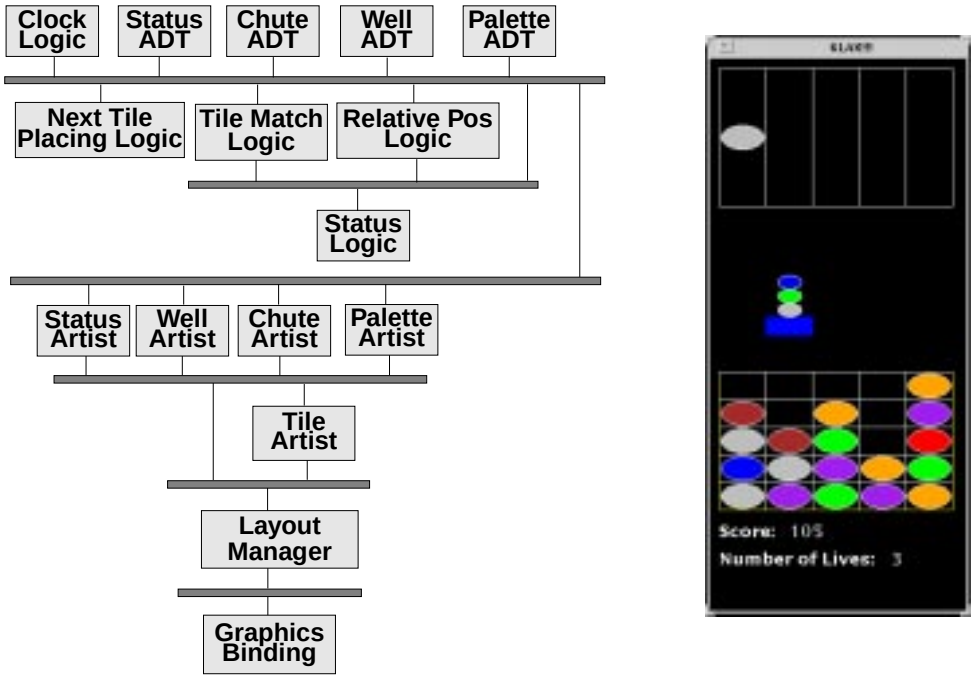
Architectural Perspectives



Example Architecture — Compiler



Example Architecture — Video Game



Analogies to Software Architecture

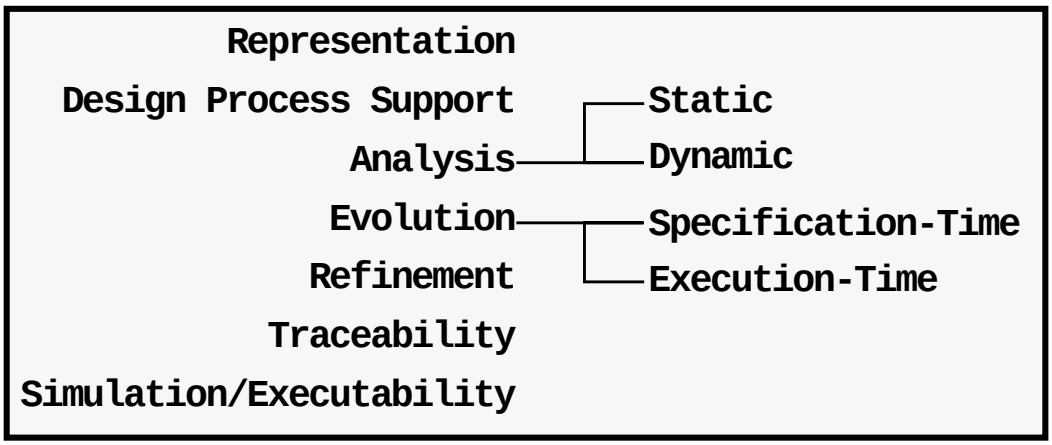
- Hardware architecture
  - small number of design elements
  - scale by replication of (canonical) design elements
- Network architecture
  - focus on topology
  - only a few topologies considered
    - e.g., star, ring, grid
- Building architecture
  - multiple views
  - styles

Current Treatment of Software Architectures

- Understood at the level of intuition, anecdote, and folklore
- Informal descriptions
  - boxes and lines
  - informal prose
- Semantically rich vocabulary that conveys a lot
  - RPC
  - client-server
  - pipe and filter
  - layered
  - distributed
  - OO
- Is this level of informality really a critical problem?

What Are Software Architectures Used for?

- Architectural domains
  - classes of problems or areas of concern in architecture



Representation	Design Process Support	Analysis	Evolution	Refinement	Traceability	Simulation / Executability

Representation

- Principal problems
  - aid stakeholder communication and understanding
- Desired solutions
  - multiple perspectives
- Achievable via
  - graphical notations
  - additional views: control flow, data flow, process, resource utilization
  - explicit configuration modeling

Representation	Design Process Support	Analysis	Evolution	Refinement	Traceability	Simulation / Executability

Design Process Support

- Principal problems
  - (de)compose large, distributed, heterogeneous systems
- Desired solutions
  - multiple perspectives
  - design guidance and rationale
- Achievable via
  - active support for specification
    - proactive vs. reactive
    - non-intrusive vs. intrusive

Representation	Design Process Support	Analysis	Evolution	Refinement	Traceability	Simulation / Executability

Static Analysis

- Principal problems
  - evaluate system properties upstream to reduce number and cost of errors
  - architecture is analyzed without executing it
- Desired solutions
  - internal consistency
  - concurrent and distributed properties
  - design heuristics and style rules
- Achievable via
  - parsers, compilers, model checkers
  - schedulability and resource utilization
  - design critics

Representation	Design Process Support	**Analysis**	Evolution	Refinement	Traceability	Simulation / Executability

Dynamic Analysis

- Principal problems
  - same as static analysis
  - architecture is analyzed during execution
    - → how do you execute an architecture?
- Desired solutions
  - testing and debugging
  - assertion checking
  - specification and checking of important runtime properties
- Achievable via
  - scenarios
  - discovering properties through simulation
  - event visualization and filtering

Representation	Design Process Support	Analysis	**Evolution**	Refinement	Traceability	Simulation / Executability

Specification-Time Evolution

- Principal problems
  - evolution of design elements, systems, and system families
- Desired solutions
  - architectural equivalent of subtyping/refinement
  - incremental specification
  - system families
- Achievable via
  - heterogeneous, flexible subtyping mechanisms
  - explicit and flexible connectors
  - explicit specification of application family

Representation	Design Process Support	Analysis	Evolution	Refinement	Traceability	Simulation / Executability

Execution-Time Evolution

- Principal problems
  - same as specification-time evolution
  - must be accomplished during system execution
- Desirable solutions
  - replication, insertion, removal, and reconnection
  - planned or unplanned
  - constraint satisfaction
- Achievable via
  - constrained and unconstrained (“pure”) dynamism
  - conditional configuration
  - replication
  - analysis of architecture during system modification

Representation	Design Process Support	Analysis	Evolution	Refinement	Traceability	Simulation / Executability

Refinement

- Principal problems
  - bridge the gap between informal diagrams and programming languages
- Desired solutions
  - specify architectures at different abstraction levels
  - correct and consistent refinement across levels
- Achievable via
  - correctness-preserving mappings
  - comparative simulations of mapped architectures

Traceability

- Principal problems