Software Engineering in a Nutshell

- Multi-person construction of multi-version software
- Software Programming ≠ Software Engineering

- single developer
- "toy" applications
- short lifespan
- single or few stakeholders
- one-of-a-kind systems
- minimal maintenance

- developer teams
- complex systems
- long, indefinite lifespan
- multiple stakeholders
- system families
- maintenance is 60+% of total development costs

- multiple roles
- "toy" applications
- long, indefinite lifespan
- developer ≠ user
- system families
- maintenance is 60+% of total development costs

- "toy" applications
- multiple roles
- "toy" applications
- multiple stakeholders
- system families
- maintenance is 60+% of total development costs

Software Engineering

- Software Engineering is a Unique Brand of Engineering

- Software is malleable
- Software construction is human-intensive
- Software is intangible
- Software problems are unprecedentedly complex
- Software directly depends upon the hardware
- Software solutions require unusual rigor
- Software has discontinuous operational nature

Software Engineering “Axioms”

- Adding developers to a project will likely result in further delays and accumulated costs
- Basic tension of software engineering
  - better, cheaper, faster — pick any two!
  - functionality, scalability, performance — pick any two!
- The longer a fault exists in software
  - the more costly it is to detect and correct
  - the less likely it is to be properly corrected
- Up to 70% of all faults detected in large-scale software projects are introduced in requirements and design
  - detecting the causes of those faults early may reduce their resulting costs by a factor of 100 or more

Enter Software Architectures

- 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
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 \}
  - WHAT
  - HOW
  - WHY

- 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 — 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
  - an architecture consisting of composite components describes a system of systems
Key Architectural Concepts — 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

Key Architectural Concepts — Configurations

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

Example Architecture — Compiler

Sequential

- Lexer
- Parser
- Semantor
- Optimizer
- Code Generator

Parallel

- Lexer
- Parser
- Semantor

Internal Representation

Example Architecture — Video Game

- Clock Logic
- Status ADT
- Chute ADT
- Well ADT
- Palette ADT
- Next Tile Placing Logic
- Tile Match Logic
- Relative Pos Logic
- Status Logic
- Status Artist
- Well Artist
- Chute Artist
- Palette Artist
- Tile Artist
- Layout Manager
- Graphics Binding
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

What Are Software Architectures Used for?

<table>
<thead>
<tr>
<th>Representation</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Process Support</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Analysis</td>
<td>Specification-Time</td>
</tr>
<tr>
<td>Evolution</td>
<td>Execution-Time</td>
</tr>
<tr>
<td>Refinement</td>
<td>Traceability</td>
</tr>
<tr>
<td>Simulation/Executability</td>
<td></td>
</tr>
</tbody>
</table>

Scope of Software Architectures

- Every system has an architecture
- Details of the architecture are a reflection of system requirements and trade-offs that made to satisfy them
- Possible decision factors
  - performance
  - compatibility with legacy software
  - planning for reuse
  - distribution profile
    - current and future
  - safety, security, fault tolerance
  - evolvability
- Critical question:
  >> How does one arrive at the architecture that satisfies the requirements?

Sources of Architecture (1)

- Architecture comes from “black magic, people having ‘architectural visions’”
- Three main sources of architecture
  - theft
  - method
  - intuition
- Their ratio varies according to
  - architects’ experience
  - system’s novelty
Sources of Architecture (2)

- Theft
  - from previous similar systems
  - from literature
- Method
  - systematic and conscious
  - possibly documented
  - architecture is derived from requirements via transformations and heuristics
- Intuition
  - “the ability to conceive without conscious reasoning”
  - increased reliance on intuition increases the risk
- Routine design vs. innovative design

Routine Design

- Method is critical
  - an architecture built with 50% theft and 50% intuition is doomed to fail
- Standardized methods
- Similarity to previous solutions
- Theft
- Cheaper but not optimal
- Can be done by good designers
- Potential pitfall
  >> over-reusing

Innovative Design

- Raw invention
- Intuition
- Derivation from abstract principles
- More optimal
- More expensive
- Must be done by great designers
- Potential pitfall
  >> reinventing the wheel

Software “Architecting”

- The “architecting” problem lies in
  - decomposition of a system into constituent elements
  - composition of (existing) elements into a system
- Two idealized approaches
  - top-down
    - decompose the large problem into sub-problems
    - implement or reuse components that solve the sub-problems
  - bottom-up
    - implement new or reuse existing stand-alone components
    - compose (a subset of) the components into a system
- A realistic approach will require both
Domain-Specific Software Architectures (DSSA)

- Development in specific domains can be optimized
  - maximize method and theft
  - minimize intuition
- Reuse in specific domains is most realistic
  - reuse in general is too difficult to achieve
  - focus on particular classes of applications with similar characteristics
- Criteria for successful reuse [Biggerstaff]
  - well-understood domain
  - slowly changing
  - has inter-component standards
  - provides economies of scale
  - fits existing infrastructure

What Is DSSA?

- DSSA is an assemblage of software components
  - specialized for a particular type of task (domain)
  - generalized for effective use across that domain
  - composed in a standardized structure (topology) effective for building successful applications
    - Rick Hayes-Roth, 1994
- DSSA is comprised of
  - a domain model,
  - reference requirements,
  - a reference architecture (expressed in an ADL),
  - its supporting infrastructure/environment, and
  - a process/methodology to instantiate/refine and evaluate it.
    - Will Tracz, 1995

Architectural Style

- Architectural styles are recurring organizational patterns and idioms.
  - Shaw & Garlan
- Established, shared understanding of common design forms is a mark of a mature engineering field.
  - Shaw & Garlan
- Architectural style is an abstraction of recurring composition and interaction characteristics of a set of architectures.
  - Taylor
- Styles are key design idioms that enable exploitation of suitable structural and evolution patterns and facilitate component, connector, and process reuse.
  - Medvidovic
Basic Properties of Styles
- A vocabulary of design elements
  - component and connector types
  - e.g., pipes, filters, objects, servers
- A set of configuration rules
  - topological constraints that determine allowed compositions of elements
  - e.g., a component may be connected to at most two other components
- A semantic interpretation
  - compositions of design elements have well-defined meanings
- Possible analyses of systems built in a style
  - code generation is a special kind of analysis

Benefits of Styles
- Design reuse
  - well-understood solutions applied to new problems
- Code reuse
  - shared implementations of invariant aspects of a style
- Understandability of system organization
  - a phrase such as “client-server” conveys a lot of information
- Interoperability
  - supported by style standardization
  - e.g., CORBA, JavaBeans
- Style-specific analyses
  - enabled by the constrained design space
- Visualizations
  - style-specific depictions matching engineers’ mental models

Some Common Architectural Styles
- “Basic” styles
  - pipe and filter
  - object-oriented
  - layered
  - blackboard
  - state transition
  - client-server
  - peer-to-peer
    - event-based (a.k.a. implicit invocation)
    - push-based
- “Derived” styles
  - GenVoca
  - C2

Role of Style
- Different styles result in
  - different architectures
  - architectures with greatly differing properties
- A style does not fully influence the resulting architecture
  - a single style can result in different architectures
  - considerable room for individual judgement
  - variations among architects
  - different emphases
    - e.g., imposed by the customer
- A style defines a domain of discourse
  - about a problem (domain)
  - about the resulting system
  - different architectures lead architects to ask different questions
Example Architecture: Key Word In Context

Hello World
I am Yoda

Hello World
World Hello
I am Yoda
Yoda I am

KWIC Architectures

Software 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
- Each connector provides
  - interaction duct(s)
  - transfer of control and/or data
Reasons for Treating Connectors Independently

- Connector ≠ Component
  - components provide application-specific functionality
  - connectors provide application-independent interaction mechanisms
- Interaction abstraction and/or parameterization
- Specification of complex interactions
  - binary vs. n-ary
  - asymmetric vs. symmetric
  - interaction protocols
- Localization of interaction definition
- Extra-component system (interaction) information
- Component independence
- Connector independence
- Component interaction flexibility

Benefits of First-Class Connectors

- Separate computation from interaction
- Minimize component interdependencies
- Support software evolution
  - at component-, connector-, and system-level
- Potential for supporting dynamism
- Facilitate heterogeneity
- Become points of distribution
- Aid system analysis and testing

Connector Types

- Procedure Call
- Data Access
- Event
- Stream
- Linkage
- Distributor
- Arbitrator
- Adaptor

Connectors as Facilitators of Distribution and Heterogeneity

Diagram showing multiple components and connectors.