13 Software Architecture

- Readings

Software Architecture Terminology

- System Architecture
  - Structure: Several computers, networks, data bases, etc. connected together
    - Analogy: Plan of city
- Software Architecture (conceptual/abstract)
  - Conceptual structure: Large piece of software with many parts, interconnections
    - Analogy: Blue print of house
- Software Design (concrete)
  - Actual structure: Large piece of software with many parts
    - Analogy: Actual structure of house
- Architectural Style
  - Form of structure, e.g., "Pipes" between components, or "Layered" system, or "Bulletin board" system
    - Analogy: Style of a building

From: Rick Holt - Waterloo
Software Architecture Terminology

- Reference Architecture
  - General architecture for an application domain
  - Example: Common structure for compilers or for operating systems

- Product Line Architecture (PLA)
  - Architecture for a line of similar software products
  - Example: Software structure for a family of computer games

Models and Views

Understanding Software Architecture

- Architecture
  - In the head(s) of software developer(s), the "software architecture" may be abstract or mostly concrete
  - Is a "mental model", "wetware"; may be fuzzy, inaccurate, incomplete, incorrect

- Complexity
  - Architecture simplifies the system, by concentrating on structure, not content or semantics
  - Cognitive complexity: how hard to understand or visualize

- Reverse Engineering
  - Extraction of design (or architecture) from implementation and from developers
  - “Design recovery”
Software Architecture

- architecture of a system describes its gross structure and illuminates the top level design decisions
  - how the system is composed of interacting parts
  - the main pathways of interaction
  - the key properties of the parts
- allows high-level analysis and critical appraisal and serves as a bridge between requirements and implementation
  - an abstract description of a system, exposes certain properties, while hiding others.
- useful for:
  - understanding
  - reuse
  - construction
  - evolution
  - analysis
  - management

Software Architectures & Design

- Architectural taxonomy ("boxology")
- Architectural patterns & idioms
- Design patterns & idioms
- Reuse
  - Class libraries
  - Components
  - Frameworks
  - Middleware
Abstraction techniques in CS

- Programming Languages
  - machine language
  - symbolic assemblers
  - macro processors
  - early high-level languages
    - Fortran
      - data types served primarily as cues for selecting the proper machine instructions
    - Algol and its successors
      - data types serve to state the programmer's intentions about how data should be used.
  - later high-level languages
    - separation of a module's specification from its implementation
    - introduction of abstract data types.

- ADT
  - the software structure (which included 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)
Defns of Software Architecture

- Perry and Wolf
  - Software Architecture = {Elements, Form, Rationale}
- Shaw and Garlan
  - Software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on those patterns
- Kruchten
  - Architecture deals with abstraction, decomposition, composition, style and aesthetics.
- Canonical Building Blocks
  - Components, Connectors, Configurations

elements, form, rationale, views

architecture=
- elements
  - processing
  - data
  - connectors
- form
  - rules which constrain element placement
  - style/design
- rationale
  - selection of form
  - links to reqmnts & design
  - functional/non-functional attributes

Process View

Data View
Components

- A component is a unit of computation or a data store.
- Components are loci of computation and state.
  - Clients
  - Servers
  - Databases
  - Filters
  - Layers
  - Abstract Data Types (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.

Scope of Software Architectures

- Details of the architecture are a reflection of system requirements and trade-offs made to satisfy them, e.g.,
  - Performance
  - Compatibility with legacy software
  - Planning for reuse
  - Distribution profile
    - Current and Future
  - Safety, Security, Fault tolerance requirements
  - Evolvability Needs
    - Changes to processing algorithms
    - Changes to data representation
    - Modifications to the structure/functionality
Expected Benefits

- **Requirements**: • Clarify intentions
  • Make decisions and implications explicit
  • Permit system level analysis

- **Architecture**: • Reduce maintenance costs, directly and indirectly

- **Design**: • "The ability to conceive without conscious reasoning."
  • Increased reliance on intuition increases the risk
  • Systematic and conscious
  • Possibly documented

- **Code Integration**: • Architecture is derived from requirements via transformations and heuristics
  • From previous similar systems
  • From literature

- **Integration**: • Test & debug

- **Test Accept**: • Define/analyze change

- **Maintenance**: • Trace logic
  • Implement change

---

**Sources of Architecture**

- Architecture comes from "3 + 1" main sources:
  - Intuition, people having 'architectural visions'
  - Method
  - Theft (i.e., reuse)
  - Blind luck & black magic

- Their ratio varies according to:
  - Architect's experience
  - System's novelty

---

© Rick Adrion 2005 (except where noted)
Risk

- Routine Design - “Theft”
  - Method is critical
    - “An architecture built with 50% theft and 50% intuition is doomed to fail.”
  - Standardized methods, similar to previous solutions
  - Often cheaper, but not optimal
  - Can be done by merely “good” designers
  - Potential pitfall
    - Over-reusing

- Innovative Design - “Intuition”
  - Raw invention
    - Derivation from abstract principles
    - More optimal & more expensive
    - Must be done by “great” designers
    - Potential pitfall
      - Reinventing the wheel
      - Single point of failure in staffing

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.
Issues in Decomposition

- How do we arrive at:
  - Components?
  - Connectors?
  - Their Configuration?
- What is the adequate component granularity level?
- What constraints on components are imposed by:
  - Functional requirements
  - Non-functional requirements
  - Envisioned evolution patterns
  - System Scale
  - Computing Environment
  - Customers/Users
- What assumptions can components make about one another?

From Nenad Medvidovic University of Southern California

Issues in Decomposition

- How do components interact?
- What are the connectors in the system?
- What is the role of the connectors?
  - Mediation
  - Coordination
  - Communication
- What is the nature of the connectors?
  - Type of interaction
  - Degree of concurrency
  - Degree of information exchange

From Nenad Medvidovic University of Southern California
Issues in Composition

- Where does one locate existing:
  - Components?
  - Connectors?
  - Configurations?
- How do we determine which elements are needed?
  - Both at development time and at reuse time
- What is the adequate element granularity level?
- How do we ensure effective composition of heterogeneous elements?
- How do we know that we have the needed system?

From Nenad Medvidovic University of Southern California

Defns - Architectural Styles

- 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

From Nenad Medvidovic University of Southern California
Basic Properties of Styles

- vocabulary of design elements
- component and connector types
  - e.g., pipes, filters, objects, server
- 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
- 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

From Nenad Medvidovic University of Southern California

Categories of Styles

- idioms & patterns
  - deal with global organizational structures
- application-domain independent:
  - pipe and filter
  - client-server
  - Blackboard
  - layered
- reference models
  - specific configurations for certain application areas
  - may be effective outside their initial domains
  - e.g., canonical compiler architecture, other DSSAs

From Nenad Medvidovic University of Southern California
Components and connectors
- primary building blocks of architectures
- abstractions used by designers in defining their architectures
- most of these elements are ultimately implemented in terms of processes (as defined by the operating system) and procedure calls (as defined by the programming language).

Control issues
- Topology
  - geometric form of the control flow for the system: linear (non-branching), acyclic, hierarchical, star, arbitrary
- Synchronicity
  - interdependency of the component control states: lockstep (sequential or parallel), synchronous, asynchronous, opportunistic
- Binding time
  - time the identity of a partner in a transfer-of-control operation is established: write (i.e., source code) time, compile time, invocation time, run time

Data issues
- Topology
  - geometric shape of the system’s data flow graph: linear (non-branching), acyclic, hierarchical, star, arbitrary
- Continuity
  - the flow of data throughout the system: continuous, sporadic, high-volume (in data-intensive systems), low-volume (in compute-intensive systems)
- Mode
  - data is made available throughout the system: passed (object style from component to component), shared: copyout-copyin, broadcast, multicast
- Binding time
  - time identity of a partner in a data operation is established: write (i.e., source code)

Control/data interaction issues
- Shape
  - control flow and data flow topologies isomorphic
- Directionality
  - if shapes the same, does control flow in the same direction as data or the opposite direction.
- Type of reasoning
  - nondeterministic state machine theory, function composition
  - software substructure and analysis substructure should be compatible.
Boxology: dataflow

<table>
<thead>
<tr>
<th>Style</th>
<th>Constituent parts</th>
<th>Control issues</th>
<th>Data issues</th>
<th>Ctrl/data interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dataflow network [B+88]</td>
<td>data stream</td>
<td>arbitrary</td>
<td>arbitrary</td>
<td>i, r</td>
</tr>
<tr>
<td>• Acyclic [A’95]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Fanout [A’95]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Pipeline [DO90, S+88, A’95]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Unix pipes and filters [Bah86]</td>
<td>ascii stream</td>
<td>linear</td>
<td>linear</td>
<td>i</td>
</tr>
</tbody>
</table>

Data flow styles: Styles dominated by motion of data through the system, with no “upstream” content control by recipient

Key to column entries

Syntaxonomy
Binding time
Continuity

asynch (asynchronous)
i (invocation-time), r (run-time)
cont (continuous), lvol (high-volume), lvol (low-volume)
Analysis: pipes & filters*

- problem decomposition
  - advantages: hierarchical decomposition of system function
  - disadvantages: “batch mentality,” interactive apps?, design
- maintenance & reuse
  - advantages: extensibility, reuse, “black box” approach
  - disadvantages: lowest common denominator for data flow
- performance
  - advantages: pipelined concurrency
  - disadvantages: parsing/un-parsing, queues, deadlock with limited buffers

*to some extent batch

Rules of thumb for dataflow/pipes

- If your problem can be decomposed into sequential stages, consider batch sequential or pipeline architectures
- If in addition each stage is incremental, so that later stages can begin before earlier stages complete, then consider a pipelined architecture
- If your problem involves transformations on continuous streams of data (or on very long streams) consider a pipeline architecture
  - However, if your problem involves passing rich data representation, then avoid pipeline architectures restricted to ASCII
- If your system involves controlling action, is embedded in a physical system, and is subject to unpredictable external perturbation so that preset algorithms go awry, consider a closed loop architecture
**taxonomy: call/return**

- **main/sub**
  - hierarchical decomposition, single thread of control, structure implicit, correctness depends on subordinates

- **layered**
  - hides lower layers/services higher layer, upper="virtual machines"/lower =hw, kernel, scoping

- **object-oriented**
  - encapsulation, inheritance, polymorphism

---

**Analysis: call/return**

- **layers**
  - portability, modifiability, reuse
    - advantages: each layer is abstract machine, each layer interacts with ≤ 2 other layers, standard interfaces
  - performance, design
    - disadvantages: semantic feedback in UI, deep functionality, abstractions difficult, bridging layers

- **object-oriented**
  - portability, modifiability, reuse
    - advantages: decreased coupling, frameworks -> reuse
    - disadvantages: complex structure
  - performance, design
    - advantages: maps easily to "real world", inheritance, encapsulation
    - disadvantages: design harder, side effects, identity, inheritance difficult
**Taxonomy: data-centered**

- **transactional db**
  - large central data store, control via transactions

- **blackboards**
  - central shared + app-specific data representations, control via data state

---

**Rules of thumb: objects and repositories**

- If a central issue is understanding the data of the application, its management, and its representation, consider a repository or ADT architecture; if the data is long-lived focus on repositories
- If the representation of data is likely to change over the lifetime of the program, ADTs or objects can confine the changes to particular components
- If you are considering repositories and the input data is "noisy" and the execution order can not be predetermined, consider a blackboard
- If you are considering repositories and the execution order is determined by a stream of incoming requests and the data is highly structured, consider a DB system.
**Taxonomy: Independent Components**

- Communicating processes
  - Independent processes, point-point message passing, async/synch, RPC layered on top
- Event systems
  - Interface define allowable in/out events, event-procedure bindings: procedure "registration", communication by event "announcement", implicit action invocation on event, non-deterministic ordering

**Boxology: Independent Components**

<table>
<thead>
<tr>
<th>Style</th>
<th>Constituent parts</th>
<th>Control Issues</th>
<th>Data Issues</th>
<th>Ctrl/data interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Components</td>
<td>Connectors</td>
<td>Topology</td>
<td>Binding</td>
</tr>
<tr>
<td>Communicating proc [AAL86, Pa85]</td>
<td>arb</td>
<td>arb</td>
<td>seq</td>
<td>arb</td>
</tr>
<tr>
<td>One-way data flow, networks of filters</td>
<td>linear</td>
<td>linear</td>
<td>star</td>
<td>linear</td>
</tr>
<tr>
<td>Communication unexpectedly</td>
<td>star</td>
<td>sync</td>
<td>star</td>
<td>passed</td>
</tr>
<tr>
<td>Hearbeat</td>
<td>processes</td>
<td>message protocols</td>
<td>hier</td>
<td>hier</td>
</tr>
<tr>
<td>Probe/echo</td>
<td>message protocols</td>
<td>message protocols</td>
<td>hier</td>
<td>hier</td>
</tr>
<tr>
<td>Broadcast</td>
<td>arb</td>
<td>arb</td>
<td>arb</td>
<td>passed</td>
</tr>
<tr>
<td>Dynamic versioning</td>
<td>arb</td>
<td>arb</td>
<td>arb</td>
<td>passed</td>
</tr>
<tr>
<td>Replicated workers</td>
<td>star</td>
<td>sync</td>
<td>star</td>
<td>passed</td>
</tr>
</tbody>
</table>

**Key to column entries**

- **Topology**: 1-bit (multicast), 0-bit (unicast), star, linear (one-way)
- **Synchronicity**: seq (sequential, one thread of control), layer (independent parallel), sync (synchronous), async (asynchronous), app/applicable
- **Binding time**: s (source-time), t (target-time)
- **Continuity**: src (source), dst (destination)
- **Mode**: shared, passed, multicast, broadcast, recv (multicast), c/c (copy/copy-out)
### analysis

- **event systems**
  - advantages: no “hardwired names”, new objects added by registration
  - disadvantages: nameserver/"yellowpages” needed
- **portability, modifiability, reuse**
- **performance, design**
  - advantages: computation & coordination are separate objects/more independent, parallel invocations
  - disadvantages: no control over order of invocation, correctness, performance penalty from communication overhead

### Rules of thumb

- If your task requires a high degree of flexibility-configurability, loose coupling between tasks, and reactive tasks, consider interacting processes
  - If you have reason not to bind the recipients of signals to their originators, consider an event architecture
  - If the task are of a hierarchical nature, consider a replicated worker or heartbeat style
  - If the tasks are divided between producers and consumers, consider a client-server style (naive or sophisticated)
  - If it makes sense for all of the tasks to communicate with each other in a fully connected graph, consider a token-passing style
taxonomy: virtual machines

- **interpreters**
  - simulate functionality which is not native to the run-time system; execution engine “implemented” in software
- **rule-based systems**
  - specialization of an interpreter

Analysis: virtual machines

- **interpreters**
  - portability, modifiability, reuse
    - disadvantages: map into actual implementation?
  - performance, design
    - advantages: simulate non-native functionality, can simulate “disaster” modes for safety analysis
    - disadvantages: much slower than actual system, additional layer of software to be verified
- **Rules of thumb: virtual machines**
  - If you have designed a computation, but have no machine on which you can execute it, consider a virtual interpreter architecture.
Client-Server Style

- instance of a more general style
- distributed systems
- components are clients and servers
  - servers do not know the number or identities of clients
  - clients know server's identity
- connectors are RPC-based interaction protocols
- number of different flavors of client-server

Push-Based Style

- distinguished from pull-based (e.g., the Web)

- employee information systems
- stock ticker
- components are producer, receiver, channel, broadcaster, transport system (repeater, cache, proxy -- transparent to all other components)
- asymmetric communication model, #producers##receivers and fewer producers but more receivers per producer than event-based style
Heterogeneous Styles

From Nenad Medvidovic University of Southern California

Observations

- 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
Open Issues

- use of styles is generally ad-hoc
- difficult to delimit system aspects that can/should be specified by a style
- difficult to compare styles based on their properties
- difficult to relate systems developed in different styles
- difficult to select appropriate style(s) for a given problem
- unclear how existing styles can be most effectively combined to produce a new style
- what is the relationship between domains and styles?

Product Line Architecture (PLA)

- Take advantage of commonality
- Bound variability

Approaches
- Proactive
  - Develop core assets first
- Reactive
  - Start with existing projects, identify and extract core assets
- Incremental
Problem and Solution

- Problem:
  - Software architecture is too complex to be captured using a single diagram, and not all aspects of it are interesting at different moments and to different stakeholders. How to manage this complexity?

- Solution:
  - Represent different aspects and different characteristics of the architecture through multiple views.

Views

- What is a view?
  - A view is a presentation of a model, which is a complete description of a system from a particular perspective.

- Proposed views:
  - Logical View - captures the object model
  - Process View - captures the concurrency and synchronization aspects
  - Development View - captures static organization of the software in its development environment
  - Physical View - captures the way software is mapped on hardware
  - The "4+1" view: these plus scenarios
Models and Views

Logical View
- Functionality

Implementation View
- Programs
- Software management

Process View
- System integration
- Performance
- Scalability
- Throughput

Deployment View
- System engineering
- System topology
- Delivery, installation
- Communication

Use Case View

Scenarios Diagrams
- Collaboration Diagrams
- Statechart Diagrams
- State Diagrams
- Activity Diagrams
- Sequence Diagrams
- Class Diagrams
- Component Diagrams
- Object Diagrams
- Use Case Diagrams
- Deployment Diagrams
4+1 view of software architecture

end users
- functionality

programmers
- software management

logical view

development view

scenarios

process view

physical view

system integrators
- performance
- scalability
- throughput

system engineers
- system topology
- delivery
- installation
- telecommunication

The Logical Architecture

- Represented by Logical View
  - of interest to end-user
  - supports functional requirements
  - presents key abstractions mostly from the problem domain
- Class diagrams show how classes are grouped together, class’ interface (functionality) and associations
  - “close” to the Development Architecture
  - usually deduced from Scenario View (or Use-Case view)
  - many case tools support it (UML tools, E-R tools etc.)
The Process Architecture

- Represented by Process View
  - of interest to system designer, integrator
  - concerned with performance, availability, S/W fault tolerance, integrity
  - presents concurrency and distribution of processes, how abstractions from Logical View map to processes
- Components: Tasks
- Connectors: rendezvous, broadcasts,…
- Containers: process
  - “close” to the Physical Architecture
  - tool support: UNAS/SALE, DADS
example: process view

example: Alcatel PBX

controller process

controller task (low rate)

controller task (high rate)

terminal process

process

collectors

unspecified

message

style: indep.component notation: Booch (Ada tasking)

The Development Architecture

- Represented by Development View
  - of interest to developer, manager
  - concerns: organization, reuse, portability, line-of-product
  - presents actual software module organization
- subsystems organized in a hierarchy of layers
- “close” to the Logical Architecture
  - usually deduced from Logical Architecture
  - tools: Apex, SoDA
example: development view

example: Alcatel PBX

layer1: human-computer interface
        external systems

layer2: 

layer3: 

layer4: 

layer5: bindings

The Physical Architecture

- Represented by Physical View
  - of interest to system designer
  - concerns: scalability, performance, availability, reliability
  - presents how processes, objects etc. are mapped onto processing nodes
- Components: processing nodes
- Connectors: LAN, WAN, bus, ...
- Containers: Physical Subsystem
  - "close" to the Process Architecture
  - strongly influenced by Process Architecture
  - tools: UNAS, DADS
example: physical view

example: Alcatel PBX

- components
  - processor
  - other device

- connectors
  - comm line
  - comm line (non-perm)
  - uni-dir comm line
  - hi-bw comm line

style: indep.comonents notation: UNAS

Physical view (with process allocation)

example: Alcatel PBX
Scenarios

- Instances of Use-Cases, unify all views
  - of interest to end-user, developer
  - concerns: understandability
- Textual domain process descriptions, object scenario diagrams and object interaction diagrams
  - used as a driver to discover architectural elements, validation of design
  - tools: UML case tools

1. off-hook
2. dial tone
3. digit
4. digit
5. open conversation

controller  terminal  numbering plan

conversation
### UML & RUP

- **SW Development Life Cycle**
  - Use-case driven
    - use cases are used as a primary artifact for establishing the desired behavior of the system, for verifying and validating the system's architecture, for testing, and for communicating among the stakeholders of the project
  - Architecture-centric
    - a system's architecture is used as a primary artifact for conceptualizing, constructing, managing, and evolving the system under development
  - Iterative
    - one that involves managing a stream of executable releases
  - Incremental
    - one that involves the continuous integration of the system's architecture to produce these releases

- **Architectural View Mismatches in UML**
  - Different UML diagrams present different system views
  - redundant information across views
  - Key challenge is to ensure inter-view consistency

### Architecture Description Languages

- formal notations for representing and analyzing architectural designs
- provide both a conceptual framework and a concrete syntax for characterizing software architectures
- tools for parsing, displaying, compiling, analyzing, or simulating architectural descriptions.
### ADL Examples

- **Adage**
  - supports the description of architectural frameworks for avionics navigation and guidance
- **Aesop**
  - supports the use of architectural styles
- **C2**
  - supports the description of user interface systems using an event-based style
- **Darwin**
  - supports the analysis of distributed message-passing systems
- **Meta-H**
  - provides guidance for designers of real-time avionics control software;
- **Rapide**
  - allows architectural designs to be simulated, and has tools for analyzing the results of those simulations;
- **SADL**
  - provides a formal basis for architectural refinement;
- **UniCon**
  - has a high-level compiler for architectural designs that supports a mixture of heterogeneous component and connector types;
- **Wright**
  - supports the formal specification and analysis of interactions between architectural components.

### formal architectural specification.

- **module interconnection languages**
  - static aspects of component interaction
  - definition and use of types, variables, and functions among components
  - examples: INTERCOL, PIC, CORBA/IDL
- **process algebras**
  - dynamic interplay among components
  - concerned with the protocols by which components communicate
  - examples: Wright (based on CSP), Chemical Abstract Machine (based on term rewriting)
- **event languages**
  - identification and ordering of events
  - event is a very flexible, abstract notion
  - example: Rapide
Evaluation & analysis

- conduct a formal review with external reviewers
  - time the evaluation to best advantage
  - choose an appropriate evaluation technique
  - create an evaluation contract
  - limit the number of qualities to be evaluated
  - insist on a system architect
- benefits
  - financial
  - increased understanding and documentation of the system
  - detection of problems with the existing architecture
  - clarification and prioritization of requirements
  - organizational learning

Benefits

- examples
  - AT&T
    - 10% reduction in project costs, on projects of 700 staff days or longer, the evaluation pays for itself.
  - consultants
    - reported 80% repeat business, customers recognized sufficient value
  - where architecture reviews did not occur
    - customer accounting system estimated to take two years, took seven years, re-implemented three times, performance goals never met
    - large engineering relational database system, performance made integration testing impossible, project was cancelled after twenty million dollars had been spent.
Reverse Engineering

Architecture-based modeling, analysis, and evolution environment (e.g., DRADIS)

Design environment (e.g., Rational Rose®)

System generation and development environment

Class Diagram

State Transition Diagram

Sequence Diagram

Architecture in ADL

Architecture in UML

Design in UML

Implementation

Nenad Medvidovic Assessing the Suitability of UML for Modeling Software Architectures