COMPUTER SCIENCE

17- Software Architecture

Rick Adrion

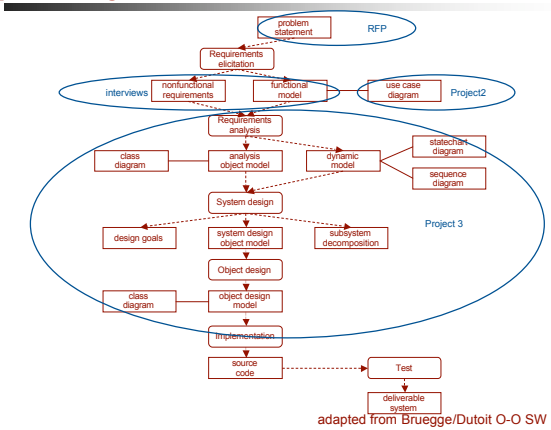
COMPUTER SCIENCE

But first

- A review of UML software development

COMPUTER SCIENCE

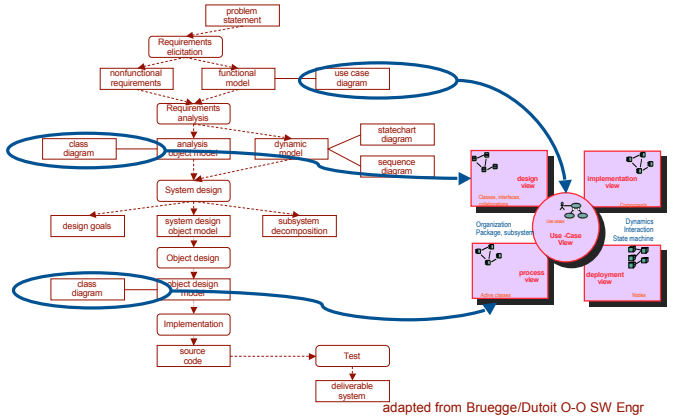
O-O System Development



adapted from Bruegge/Dutoit O-O SW Engr

COMPUTER SCIENCE

O-O System Development



adapted from Bruegge/Dutoit O-O SW Engr

Adapted from

- Various sources including:
  - David Garlan, “Software Architecture: a Roadmap,” **Proceedings of the conference on The future of Software engineering**, Limerick, Ireland, June 04 - 11, 2000
  - M. Shaw and P. Clements,”A field guide to boxology: Preliminary classification of architectural styles for software systems,” **Proceedings of COMPSAC 1997**, August 1997
  - M. Shaw and D. Garlan, Tutorial Slides on Software Architecture http://www-2.cs.cmu.edu/afs/cs/project/tinker-arch/www/html/Tutorial_Slides/Soft_Arch/quick_index.html
  - Garlan, David & Shaw, “An Introduction To Software Architecture,” Technical report, The Software Engineering Instiute, Carnegie Mellon University

Software Architecture

- architecture of a system describes its gross structure
- 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

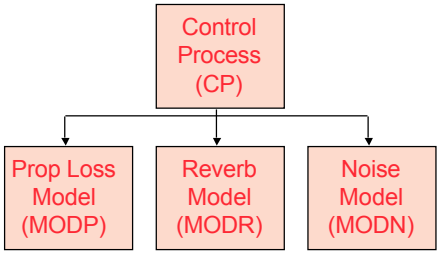
Roles of Software Architecture

- 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

Roles of Software Architecture

- Understanding:
  - simplifies the understanding of large systems using an abstraction
  - constraints on system design
  - rationale
- Construction
  - a partial blueprint for development: components and dependencies
- Evolution
  - dimensions along which a system is expected to evolve
  - "load-bearing walls" -> ramifications of changes, cost estimation
  - separate concerns about the functionality of a component from the ways in which that component is connected to (interacts with) other components
- Analysis
  - consistency checking
  - conformance
    - to constraints
    - to quality attributes
  - dependence analysis
  - domain-specific analyses for architectural styles
- Reuse
  - reuse of large components and frameworks
- Management
  - leads to a much clearer understanding of requirements, implementation strategies, and potential risks

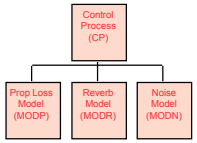
Architecture was largely ad hoc



- is this an architecture?

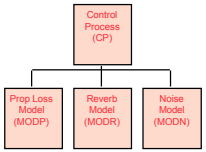
Example

- what is the nature of the components, and what is the significance of their separation?
  - do they run on separate processors?
  - do they run at separate times?
  - do the components consist of processes, programs, or both?
  - do the components represent ways in which the project labor will be divided, or do they convey a sense of runtime separation?
  - are they modules, objects, tasks, functions, processes, distributed programs, or something else?



Example

- what is the significance of the links?
  - do the links mean the components communicate with each other, control each other, send data to each other, use each other, invoke each other, synchronize with each other, or some combination of these or other relations?
- what is the significance of the layout?
  - why is CP on a separate (higher) level?
  - does it call the other three components, and are the others not allowed to call it?
  - was there simply not room enough to put all four components on the same row in the diagram?



Historically

- Architecture was largely ad hoc affair
  - Designers freely use informal patterns/idioms
    - informal with imprecise semantics
    - diagrams + prose, but no rules
  - Designers use system-level abstraction
    - overall organization (styles)
    - components and interactions
  - Designers compose systems from subsystems
    - but, tend to think statically
    - select structure by default, rather than by design
- Key events
  - Parnas recognized the importance of system families and architectural decomposition principles based on information hiding
  - Dijkstra proposed certain system structuring principles

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 it 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.

Abstraction techniques in CS

- 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)

two trends

- recognition of a shared repertoire of methods, techniques, patterns and idioms for structuring complex software systems
- concern with exploiting commonalities in specific domains to provide reusable frameworks for product families

two trends

- recognition of a shared repertoire of methods, techniques, patterns and idioms for structuring complex software systems
  - "Camelot is based on the **client-server model** and uses remote procedure calls both locally and remotely to provide communication among applications and servers."
  - "Abstraction **layering and system decomposition** provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices. The architecture encourages a **client-server model** for the structuring of applications."
  - "We have chosen a **distributed, object-oriented approach** to managing information."
  - "The easiest way to make the canonical sequential compiler into a concurrent compiler is to **pipeline** the execution of the compiler phases over a number of processors. . . . A more effective way [is to] split the source code into many segments, which are concurrently processed through the various phases of compilation [by multiple compilerprocesses] before a final, merging pass recombines the object code into a single program."

two trends

- concern with exploiting commonalities in specific domains to provide reusable frameworks for product families; examples include:
  - the standard decomposition of a compiler
  - standardized communication protocols, e.g., Open Systems Interconnection Reference Model (a layered network architecture)
  - tools, e.g., NIST/ECMA Reference Model (a generic software engineering environment architecture based on layered communication substrates)
  - fourth-generation languages
  - user interface toolkits and frameworks, e.g., X Window System (a distributed windowed user interface architecture based on event triggering and callbacks)

Why Important?

- mutual communication.
  - software architecture represents a common high-level abstraction of the system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, forming consensus, and communicating with each other.
- transferable abstraction of a system.
  - software architecture embodies a relatively small, intellectually graspable model for how the system is structured and how its components work together; this model is transferable across systems; in particular, it can be applied to other systems exhibiting similar requirements, and can promote large scale reuse.

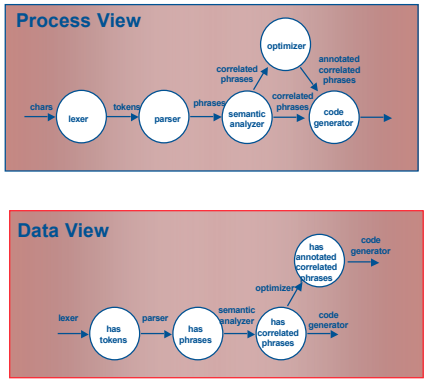
Why Important?

- early design decisions
  - software architecture represents the embodiment of the earliest set of design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its service in deployment, and its maintenance life.
- architecture
  - provides builders with constraints on implementation
  - dictates organizational structure for development and maintenance projects
  - permits or precludes the achievement of a system's targeted quality attributes
  - Helps in predicting certain qualities about a system architecture can be the basis for training
  - helps in reasoning about and managing change

elements, form, rationale, views

architecture=

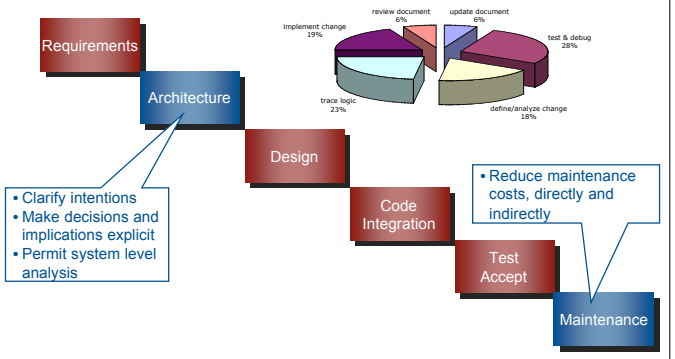
- elements
  - processing
  - data
  - connectors
- form
  - rules which constrain element placement
  - style/design
- rationale
  - selection of form
  - links to reqmnts & design
  - functional/non-functional attributes



architectural styles/idioms

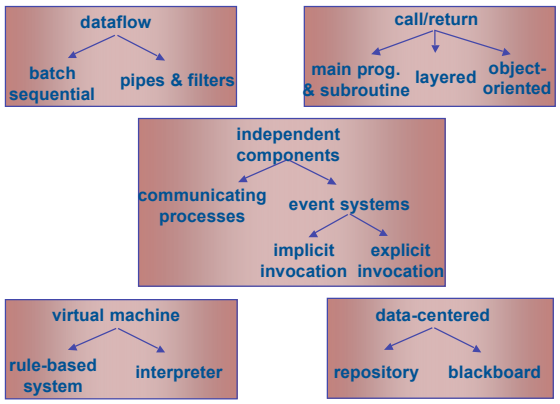
- architectural style =
  - Components: locus of computation
    - filters, databases, objects, clients, servers, ADTs
  - Connectors: mediate interactions of components
    - procedure call, pipes, event broadcast
  - Properties: specify info for construction & analysis
    - Signatures, pre/post conditions, RT specifications
- other
  - topology
  - underlying structural model?
  - underlying computational model?

Expected Benefits



© David Garlan CMU

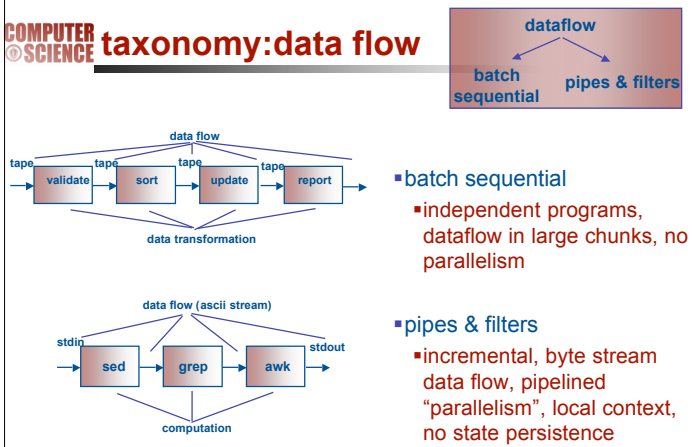
taxonomy



"Boxology"

- 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)
- Data issues
  - Mode
    - data is made available throughout the system: passed (object style from component to component), shared: copyout-copy-in, 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.

taxonomy:data flow



- batch sequential
  - independent programs, dataflow in large chunks, no parallelism
- pipes & filters
  - incremental, byte stream data flow, pipelined "parallelism", local context, no state persistence

Boxology: dataflow

Style	Constituent parts		Control issues			Data issues				Ctrl/data interaction	
	Comp-onents	Conn-ectors	Topo-logy	Synch-ronicity	Bind-ing time	Topo-logy	Contin-uity	Mode	Bind-ing time	Isomor-phic shapes	Flow dir-ections
Data flow styles: Styles dominated by motion of data through the system, with no "upstream" content control by recipient											
Dataflow network [B+88]	trans-ducers	data stream	arbi-trary	asynch	i, r	arbi-trary	cont lvol or hvol	passed	i, r	yes	same
• Acyclic [A+95]			acyclic			acyclic					
• Fanout [A+95]			hier-archy			hier-archy					
• Pipeline [DG90, Se88, A+95]			linear			linear					
-Unix pipes and filters [Ba86a]		ascii stream			i				i		
Key to column entries											
Synchronicity	asynch (asynchronous)										
Binding time	i (invocation-time), r (run-time)										
Continuity	cont (continuous), hvol (high-volume), lvol (low-volume)										

Analysis: pipes & filters*

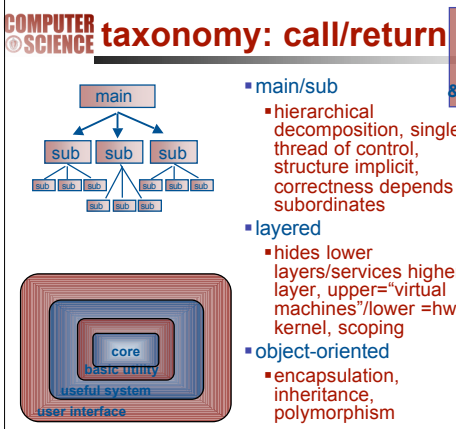
- problem decomposition
  - advantages: hierarchical decomposition of system function
  - disadvantages: "batch mentality," interactive apps?, design
- maintenance & reuse
  - advantages: extensibility, reuse, "black box" approach
  - disadvantages: lowest common denominator for data flow
- performance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

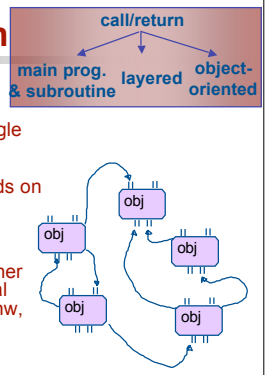
***to some extent batch**

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



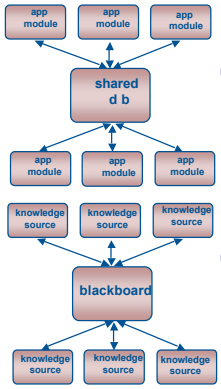


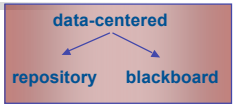
- main/sub
  - hierarchical decomposition, single thread of control, structure implicit, correctness depends on subordinates
- layered
  - hides lower layers/services higher layer, upper="virtual machines"/lower =hw, kernel, scoping
- object-oriented
  - encapsulation, inheritance, polymorphism

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, inheitance difficult

Taxonomy: data-centered



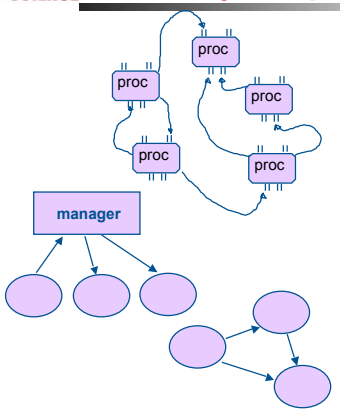


- transactional db
  - large central data store, control via transactions
- blackboards
  - central shared + app-specific data representations, control via data state

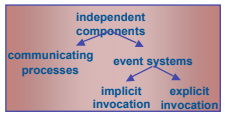
Rules of thumb: objects and repositories

- 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, asynch/synch, RPC layered on top
- event systems
  - interface define allowable in/out events, event-procedure bindings: procedure "registration", communiation by event "announcement", implicit action invocation on event, non-deterministic ordering



Boxology: independent components

Style	Constituent parts		Control issues			Data issues				Ctrl/data interaction	
	Comp-onents	Connec-tors	Topo-logy	Synch-ronicity	Bind-ing time	Topo-logy	Contin-uity	Mode	Bind-ing time	Isomor-phic shapes	Flow dir-ections
Interacting process styles: Styles dominated by communication patterns among independent, usually concurrent, processes											
Communicating processes [An91, Pa85]	processes	message protocols	arb	any but seq	w, c, r	arb	spor lvol	any	w, c, r	possibly	if iso-morphic either
One-way data flow, networks of filters			linear	asynch		linear		passed		yes	same
Client/server request/reply			star	synch		star		passed		yes	opposite
Heartbeat			hier	ls/par		hier or star		passed shared ci/co		no	same
Probe/echo			incom-plete graph	asynch		incom-plete graph		passed	w, c	yes	same
Broadcast			arb	asynch		star		bdcast		no	same
Token passing			arb	asynch		arb.		passed		yes	same
Decentralized servers											
Replicated workers			hier	synch		hier		passed shared		yes	yes
Key to column entries											
Topology	hier (hierarchical), arb (arbitrary), star, linear (one-way)										
Synchronicity	seq (sequential, one thread of control), ls/par (lockstep parallel), synch (synchronous), asynch (asynchronous), opp (opportunistic)										
Binding time	w (write-time--that is, in source code), c (compile-time), i (invocation-time), r (run-time)										
Continuity	spor (sporadic), lvol (low-volume)										
Mode	shared, passed, bdcast (broadcast), mcast (multicast), ci/co (copy-in/copy-out)										

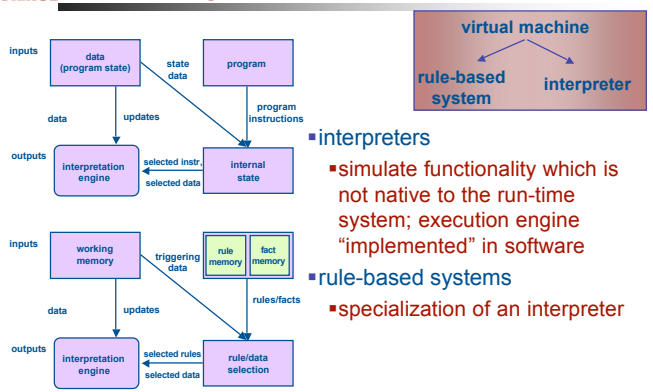
analysis

- event systems
- portability, modifiability, reuse
  - advantages: no "hardwired names", new objects added by registration
  - disadvantages: nameserver/"yellowpages" needed
- 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 (naïve 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.

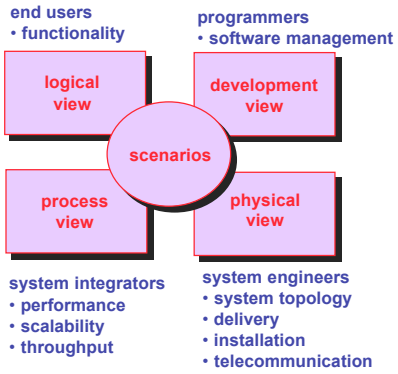
Problem and Solution

- 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

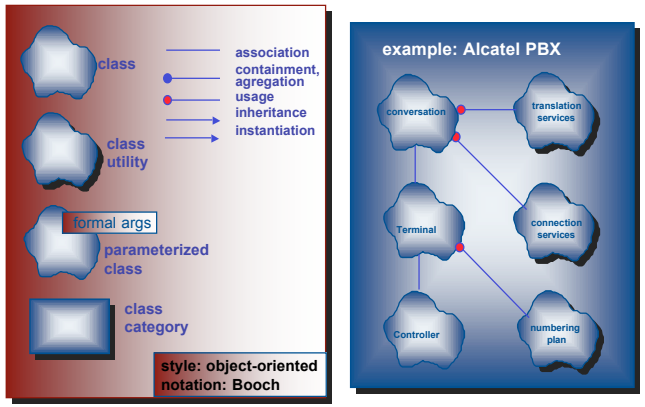
4+1 view of software architecture



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.)

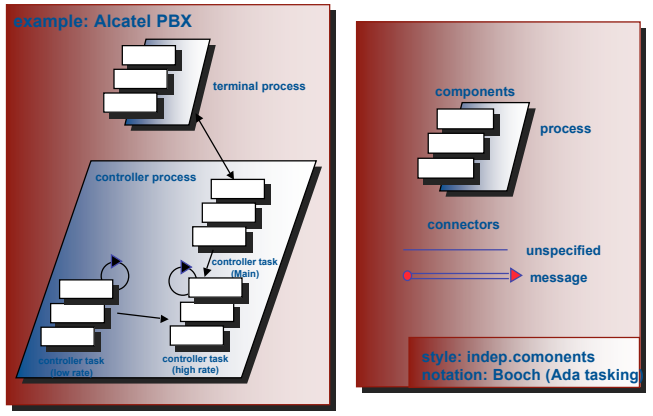
example: logical view



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

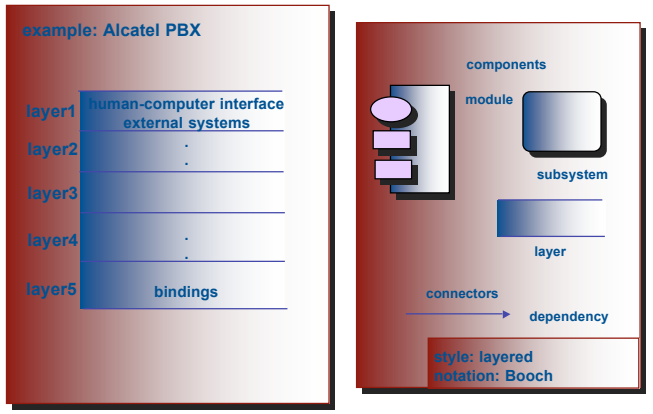
example: process view



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

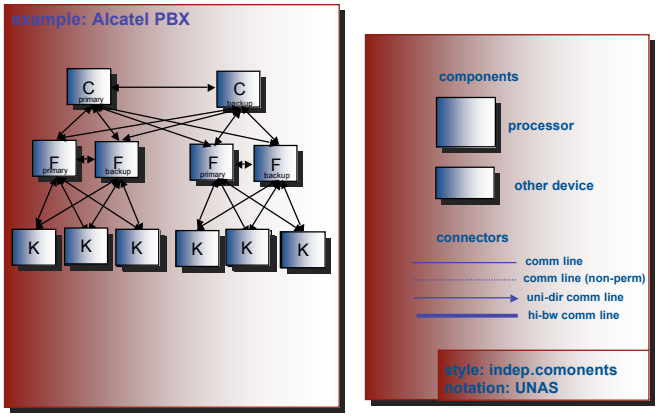
example: development view



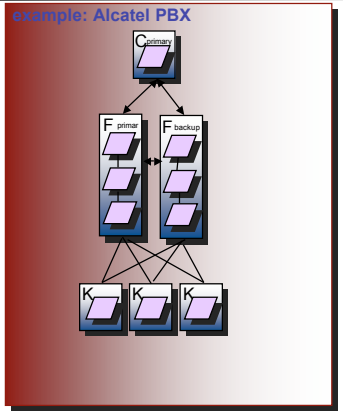
The Physical Architecture

- 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



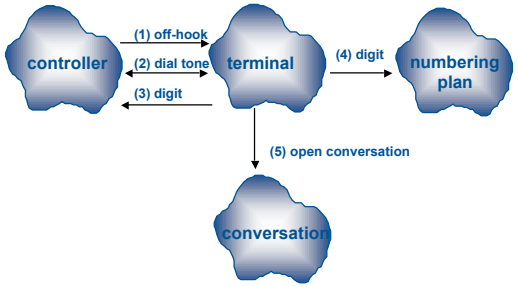
Physical view (with process allocation)



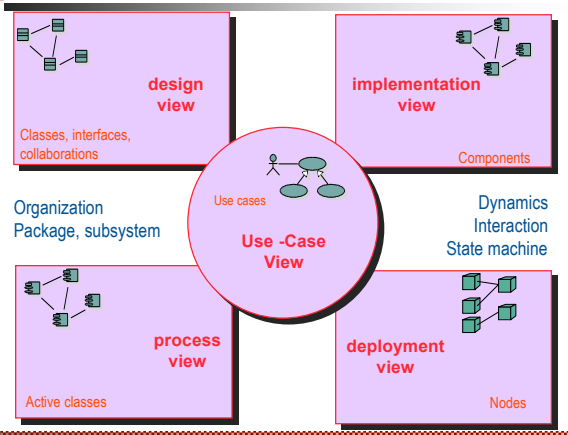
Scenarios

- 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

Scenarios



The Rational 4+1 Views



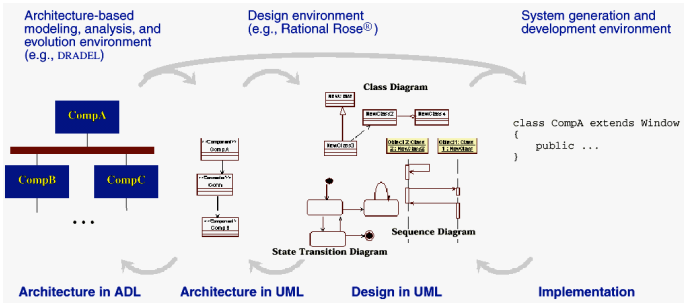
UML SW Development Life Cycle

- 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
- Ramifications on round-trip engineering

Round-Trip Software Engineering Using UML



Nenad Medvidovic Assessing the Suitability of UML for Modeling Software Architectures

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 anevent-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 sufficientvalue
  - 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.

Architecture vs Frameworks

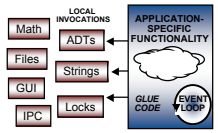
- Frameworks
  - an object-oriented reuse technique
  - used successfully for some time & are an important part of the culture of long-time object-oriented developers,
  - BUT they are not well understood outside the object-oriented community and are often misused
- Question:
  - are frameworks mini-architectures, large-scale patterns, or they are just another kind of component?
- Definitions
  - a framework is a **reusable design** of all or part of a system that is **represented by a set of abstract classes** and the way their instances interact
  - a framework is the skeleton of an application that can be customized by an application developer

Ralph E. Johnson, "Frameworks= (Components+Patterns)."**Communications of the ACM,** October 1997/Vol. 40, No. 10

Frameworks & Class Libraries

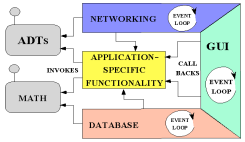
- developers often do not even know they are using a framework, but refer to a "class library"
- frameworks differ from other class libraries by reusing high-level design
  - more to learn before a class can be reused
  - can never be reused in isolation; typically a set of classes must be learned at once
- you can often tell that a class library is a framework if there are dependencies among its components and if programmers who are learning it complain about its complexity.

Frameworks & Class Libraries



Class Library Architecture

- A class is a unit of abstraction & implementation in an OO programming language



Framework Architecture

- A framework is an integrated set of abstract classes that can be customized for instances of a family of applications

Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

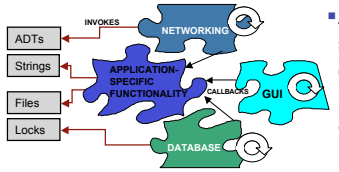
Components & frameworks

- Frameworks
  - were originally intended to be reusable components
    - but reusable O-O components have not found a market
  - are a component in the sense that
    - venders sell them as products
    - an application might use several frameworks.
  - BUT
    - they more customizable than most components
    - have more complex interfaces
      - must be learned before the framework can be used
- a component represents **code reuse**, while frameworks are a form of **design reuse**

Components & frameworks

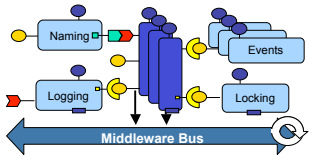
- frameworks
  - provide a reusable context for components
  - provide a standard way for components to handle errors, to exchange data, and to invoke operations on each other
    - "component systems'' such as OLE, OpenDoc, and Beans, are really frameworks that solve standard problems that arise in building compound documents and other composite objects. make it easier to develop new components
  - enable making a new component (such as a user interface) out of smaller components (such as a widget)
  - provide the specifications for new components and a template for implementing them.
- a good framework can reduce the amount of effort to develop customized applications by an order of magnitude

Frameworks & Components



Framework Architecture

- A framework is an integrated set of abstract classes that can be customized for instances of a family of applications



Component Architecture

- A component is an encapsulation unit with one or more interfaces that provide clients with access to its services

Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

Comparison

Class Libraries	Frameworks	Components
Micro-level	Meso-level	Macro-level
Stand-alone language entities	"Semi-complete" applications	Stand-alone composition entities
Domain-independent	Domain-specific	Domain-specific or Domain-independent
Borrow caller's thread	Inversion of control	Borrow caller's thread

Frameworks as Reusable Design

- Are they like other techniques for reusing high-level design, e.g., templates or schemas?
- templates or schemas
  - usually depend on a special purpose design notation
  - require special software tools
- frameworks
  - are expressed in a programming language
  - makes them easier for programmers to learn and to apply
  - no tools except compilers
  - can gradually change an application into a framework
  - because they are specific to a programming language, some design ideas, such as behavioral constraints, cannot be expressed well

Frameworks and domain-specific architectures

- A framework is ultimately an object-oriented design, while a domain-specific architecture might not be.
- A framework can be combined with a domain-specific language by translating programs in the language into a set of objects in a framework
  - window builders associated with GUI frameworks are examples of domain-specific visual programming languages
- Uniformity reduces the cost of maintenance
  - GUI frameworks give a set of applications a similar look and feel
  - using a distributed object framework ensures that all applications can communicate with each other.
  - maintenance programmers can move from one application to the next without having to learn a new design

Overview of Patterns

- Patterns
  - present solutions to common software problems arising within a certain context
  - help resolve key software design issues
    - Flexibility, Extensibility, Dependability, Predictability, Scalability,Efficiency
  - capture recurring structures & dynamics among software participants to facilitate reuse of successful designs
  - codify expert knowledge of design strategies, constraints and best practices

software patterns

- record experience of good designers
  - describe general, recurring design structures in a pattern-like format
  - problem, generic solution, usage
- solutions (mostly) in terms of O-O models
  - crc-cards; object-, event-, state diagrams
  - often not O-O specific
- patterns are generic solutions; they allow for design and implementation variations
  - the solution structure of a pattern must be “adapted” to your problem design
  - map to existing or new classes, methods, ...
    - a pattern is not a concrete reusable piece of software!

qualities of a pattern

- encapsulation and abstraction
  - each pattern encapsulates a well-defined problem and its solution in a particular domain
  - serve as abstractions which embody domain knowledge and experience
- openness and variability
  - open for extension or parametrization by other patterns so that they may work together
- generativity and composability
  - generates a resulting context which matches the initial context of one or more other patterns in a pattern language
  - applying one pattern provides a context for the application of the next pattern.
- equilibrium
  - balance among its forces and constraints

Taxonomy of Patterns & Idioms

Type	Description	Examples
Idioms	**Restricted to a particular language, system, or tool**	**Scoped locking**
Design patterns	**Capture the static & dynamic roles & relationships in solutions that occur repeatedly**	**Active Object, Bridge, Proxy, Wrapper Façade, & Visitor**
Architectural patterns	**Express a fundamental structural organization for software systems that provide a set of predefined subsystems, specify their relationships, & include the rules and guidelines for organizing the relationships between them**	**Half-Sync/Half-Async, Layers, Proactor, Publisher-Subscriber, & Reactor**
Optimization principle patterns	**Document rules for avoiding common design & implementation mistakes that degrade performance**	**Optimize for common case, pass information between layers**

Frameworks and Patterns

- frameworks represent a kind of pattern
  - e.g., Model/View/Controller is a user-interface framework often described as a pattern
  - applications that use frameworks must conform to the frameworks' design and model of collaboration, so the framework causes patterns in the applications that use it.
- frameworks are at a different level of abstraction than patterns
  - frameworks can be embodied in code, but only examples of patterns can be embodied in code.
  - a strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly
  - in contrast, design patterns have to be implemented each time they are used.

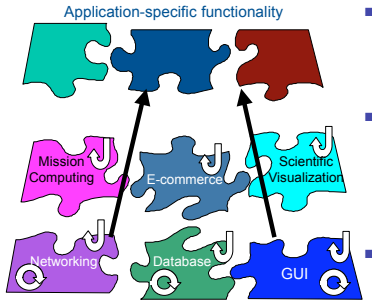
Frameworks and Patterns

- design patterns are smaller architectural elements than frameworks
  - a typical framework contains several design patterns but the reverse is never true
  - design patterns are the micro-architectural elements of frameworks.
    - e.g., Model/View/Controller can be decomposed into three major design patterns, and several less important ones
    - MVC uses the Observer pattern to ensure the view's picture of the model is up-to-date, the Composite pattern to nest views, and the Strategy pattern to cause views to delegate responsibility for handling user events to their controller.
- design patterns are less specialized than frameworks.
  - frameworks always have a particular application domain.
  - design patterns can be used in nearly any kind of application.
  - more specialized design patterns are certainly possible, even these wouldn't dictate an application architecture

Frameworks

- are firmly in the middle of reuse techniques.
- are more abstract and flexible than components,
- are more concrete and easier to reuse than a pure design (but less flexible and less likely to be applicable)
- are more like techniques that reuse both design and code, such as application generators and templates.
- can be thought of as a more concrete form of a pattern
  - patterns are illustrated by programs, but a framework is a program

Framework Characteristics

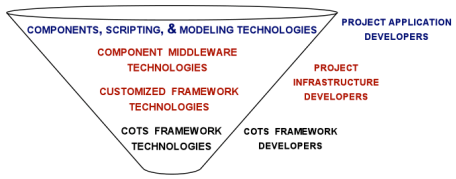


- Frameworks exhibit "inversion of control" at runtime via callbacks
- Frameworks provide integrated domain-specific structures & functionality
- Frameworks are "semi-complete" applications

Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

Using Frameworks Effectively

- Frameworks are powerful, but hard to develop & use effectively by application developers
- It's often better to use & customize COTS frameworks than to develop in-house frameworks
- Components are easier for application developers to use, but aren't as powerful or flexible as frameworks
- Successful projects are often organized using the "funnel" model

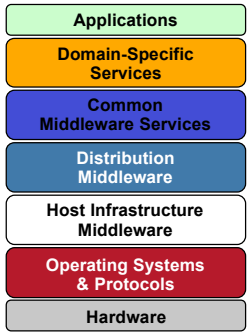


Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

Relation to Middleware

- one of the strengths of frameworks is that they are represented by traditional object-oriented programming languages.
- BUT, this is also a weakness of frameworks, however, and it is one that the other design-oriented reuse techniques do not share.
- Middleware
  - COM, CORBA, etc. address this problem, since they let programs in one language interoperate with programs in another
- Other approaches
  - some frameworks have been implemented twice so that users of two different languages can use them, such as the SEMATECH CIM framework

Evolution of Middleware



- Historically, mission-critical apps were built directly atop hardware & OS
  - tedious, error-prone, & costly over lifecycles
- There are layers of middleware, just like there are layers of networking protocols
- Standards-based COTS middleware helps:
  - Control end-to-end resources & QoS
  - Leverage hardware & software technology advances
  - Evolve to new environments & requirements
  - Provide a wide array of reuseable, off-the-shelf developer-oriented services

Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

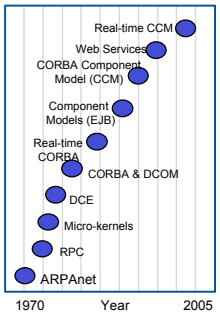
Middleware

- Infrastructure middleware.
  - encapsulates core OS communication and concurrency services to eliminate many tedious, error-prone, and non-portable aspects of developing and maintaining distributed applications using low-level network programming mechanisms, such as sockets
  - Examples: the Java Virtual Machine (JVM) and the ADAPTIVE Communication Environment (ACE).
- Distribution middleware
  - builds upon the lower-level infrastructure middleware to automate common network programming tasks, such as parameter marshaling/demarshaling, socket and request demultiplexing, and fault detection/recovery
  - Examples: Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA), Microsoft's Distributed COM (DCOM), and JavaSoft's Remote Method Invocation (RMI).

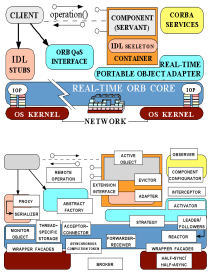
Middleware

- Common middleware services
  - augments the distribution middleware by defining domain-independent services, such as event notifications, logging, multimedia streaming, persistence, security, transactions, fault tolerance, and distributed concurrency control
  - applications can reuse these services to perform common distribution tasks that would otherwise be implemented manually.
- Domain-specific Services
  - tailored to the requirements of particular domains, such as telecommunications, e-commerce, health-care, or process automation
  - are generally reusable, and thus are the least mature of the middleware layers today
  - embody domain-specific knowledge, however, they have the most potential to increase system quality and decrease the cycle-time and effort required to develop particular types of networked applications

Progress



- significant progress in QoS-enabled middleware, stemming in large part from the following trends:
  - years of iteration, refinement, & successful use
  - maturation of middleware standards
    - .NET, J2EE, CCM
    - Real-time CORBA
    - Real-time Java
    - SOAP & Web Services
  - maturation of component middleware frameworks & patterns



Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"