## 13 - Software Architecture

- 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 Institute, Carnegie Mellon University



## First, back to Design Overview

- High-Level Design
  - Components & Connections
- Low-Level Design
  - Representation & Algorithms
- Very-Low-Level Design
  - Naming, Constructs, etc.



## Modular Decomposition

- How to define the structure of a modular system?
  - A module is a well-defined component of a software system
  - A module is part of a system that provides a set of services to other modules
- What are desirable properties of a decomposition?
  - Cohesion
  - Coupling
  - Complexity
  - Correctness
  - Correspondence
- Strategies for decomposition
  - Information Hiding
  - Layering



## Information hiding

- perhaps the most important intellectual tool developed to support software design; makes anticipation of change a centerpiece in decomposition into modules
- are OO & IH the same?
  - OO classes are chosen based on the domain of the problem (in most OO analysis approaches), not necessarily based on change, but they are obviously related (e.g., separating interface from implementation)
- Notkin's IH "Central Premises"
  1. can effectively anticipate changes
  2. changing an implementation is the best change, since it's isolated
  3. semantics of a module must remain unchanged when implementations are replaced
  4. one implementation can satisfy multiple clients
  5. information hiding can be recursively applied

## Abstract Modules

- Abstract objects
  - Objects manipulated via interface functions
  - Data structure hidden to clients
- Abstract data types
  - Many instances of abstract objects may be generated



## Abstract objects

- Examples
  - calculator of expressions expressed in Polish postfix form: a*(b+c) ◊ abc+*
  - a stack where the values of operands are shifted until an operator (assume only binary operators) is encountered in the expression

```
Interface of the abstract object STACK
    exports
    procedure PUSH (VAL: in integer);
    procedure POP_2 (VAL1, VAL2: out
```

  - How does the design anticipate change in type of expressions to be evaluated?
    - e.g., it does not adapt to unary operators



## Abstract data types (ADTs)

- Example: stack ADT

```
module  STACK_HANDLER
exports
        type  STACK = ?;
        This is an abstract data -type module; the data structure
        is a secret hidden in the implementation part.
        procedure   PUSH (S:  in out  STACK ; VAL:  in  integer);
        procedure   POP (S: in out  STACK ; VAL:  out  integer);
        function  EMPTY (S: in  STACK) : BOOLEAN;
        :
end  STACK_HANDLER
```

indicates that details of the data structure are hidden to clients

- ADTs correspond to Java and C++ classes & may also be implemented by Ada private types and Modula-2 opaque types



## Abstract data types (ADTs)

- Aother example: simulation of a gas station

```
module FIFO_CARS
uses CARS
exports
        type QUEUE : ?;
        procedure ENQUEUE (Q: in out QUEUE ; C: in CARS);
        procedure DEQUEUE (Q: in out QUEUE ; C: out CARS);
        function IS_EMPTY (Q: in QUEUE) : BOOLEAN;
        function LENGTH (Q: in QUEUE) : NATURAL;
        procedure MERGE (Q1, Q2 : in QUEUE ; Q : out QUEUE);
        This is an abstract data-type module representing
        queues of cars, handled in a strict FIFO way;
        queues are not assignable or checkable for equality,
        since ":=" and "=" are not exported.
        …
end FIFO_CARS
```

## Generic modules

- parametric with respect to a type

```
generic module GENERIC_STACK_2
. . .
exports
procedure PUSH (VAL : in T);
procedure POP_2 (VAL1, VAL2 : out T);
…
end GENERIC_STACK_2
```

- specify that a type and also an operation must be provided parameters

```
generic module M (T) with OP(T)
uses ...
        ...
end M
```

- instantiation syntax:

```
module INTEGER_STACK_2 is GENERIC_STACK_2 (INTEGER)
module M_A_TYPE is M(A_TYPE)  PROC(M_A_TYPE)
```

More on genericit



## 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

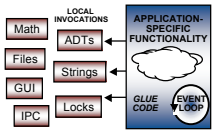
- Understanding:
  - simplifies the understanding of large systems using an abstraction
  - constraints on system design
  - rationale
- Construction
  - a partial blueprint for development: components and dependencies
- Evolution
  - dimensions along which a system is expected to evolve
  - "load-bearing 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

## Software Architectures

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

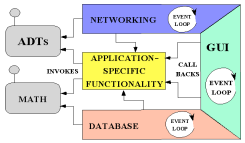



## Frameworks & Class Libraries




Class Library Architecture

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



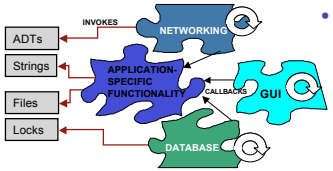

Framework Architecture

- 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"

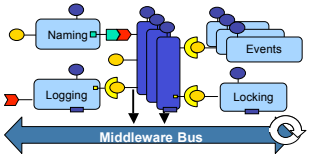


## Frameworks & Components




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

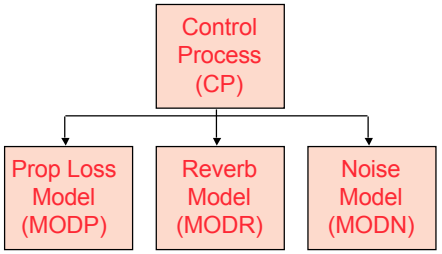
Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"



## 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 |

## Architecture was largely ad hoc

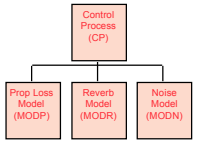



- is this an architecture?



## Example

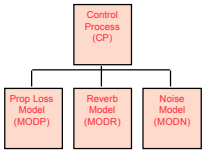
- what is the nature of the components, and what is the significance of their separation?
  - do they run on separate processors?
  - do they run at separate times?
  - do the components consist of processes, programs, or both?
  - do the components represent ways in which the project labor will be divided, 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?

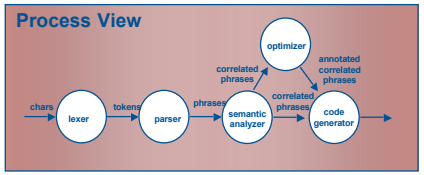
- early design decisions
  - software architecture represents the embodiment of the earliest set of design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its service in deployment, and 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

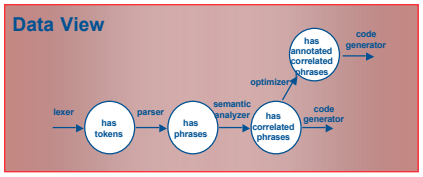


## elements, form, rationale, views

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



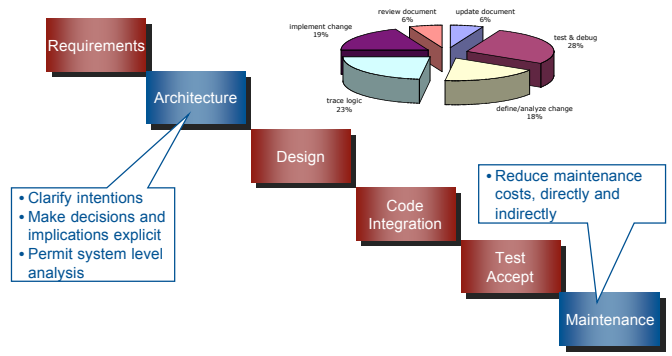



## architectural styles/idioms

- 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?



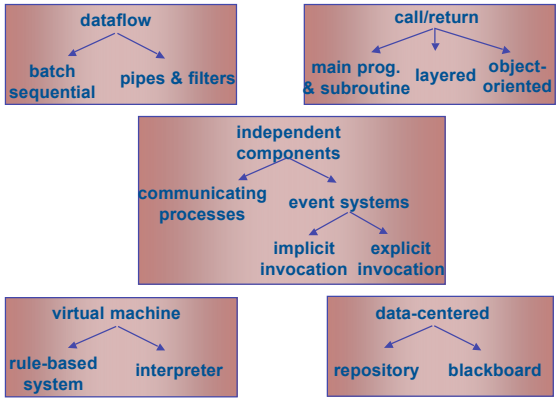
## Expected Benefits




© David Garlan CMU



## taxonomy

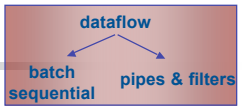


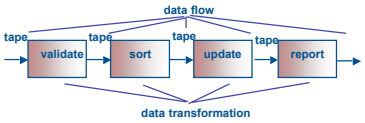



## "Boxology"

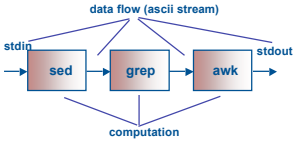
- Components and connectors
  - primary building blocks of architectures
  - abstractions used by designers in defining their architectures
  - most of these elements are ultimately implemented in terms of processes (as defined by the operating system) and procedure calls (as defined by the programming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

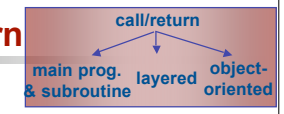
***to some extent batch**

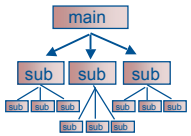


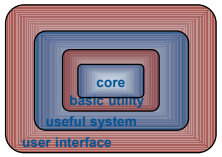
## Rules of thumb for dataflow/pipes

- 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

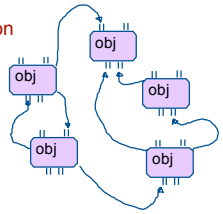
## taxonomy: call/return




- 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



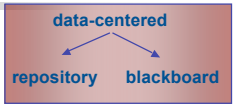


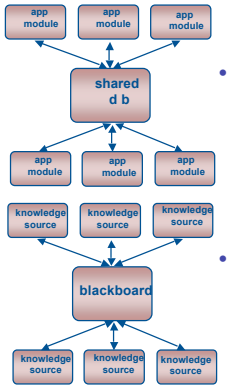

## Analysis: call/return

- 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



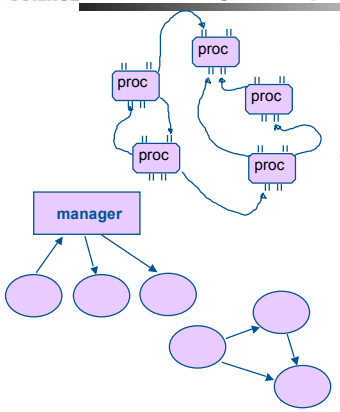

- transactional db
  - large central data store, control via transactions
- blackboards
  - central shared + app-specific data representations, control via data state



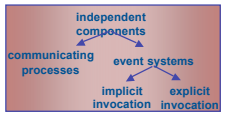
## Rules of thumb: objects and repositories

- 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 | w, c | 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 | | 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

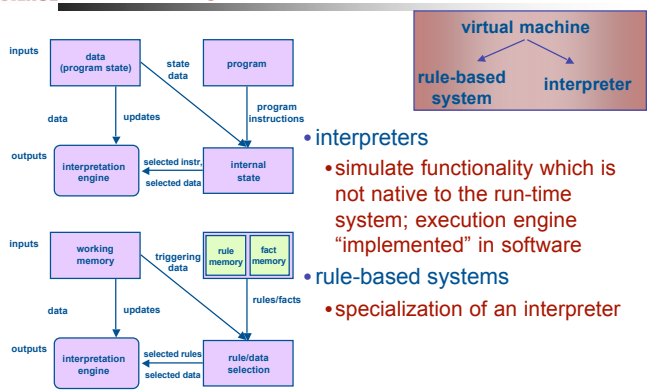
- event systems
- portability, modifiability, reuse
  - advantages: no "hardwired names", new objects added by registration
  - disadvantages: nameserver/"yellowpages" needed
- performance, design
  - advantages: computation & coordination are separate objects/more independent, parallel invocations
  - 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

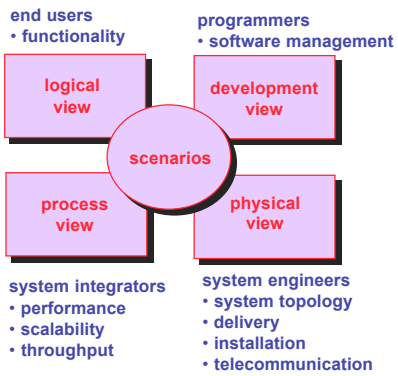
- Problem:
  - Software architecture is too complex to be captured using a single diagram, and not all aspects of it are interesting at different moments and to different stakeholders. How to manage this complexity?
- Solution:
  - Represent different aspects and different characteristics of the architecture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



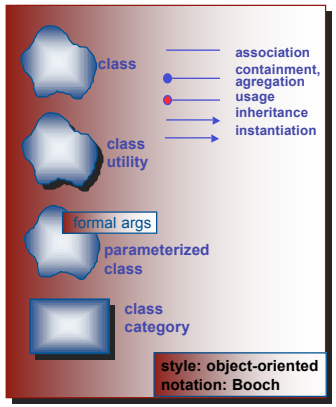


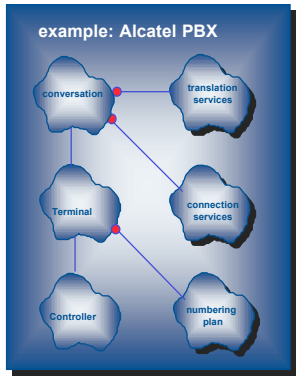

## The Logical Architecture

- 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



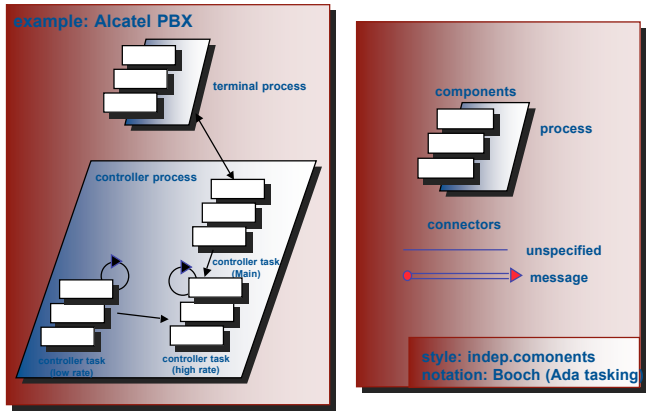






## The Process Architecture

- 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



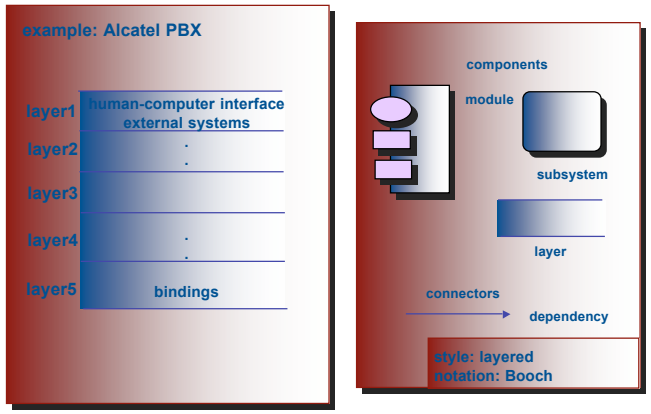



## The Development Architecture

- 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

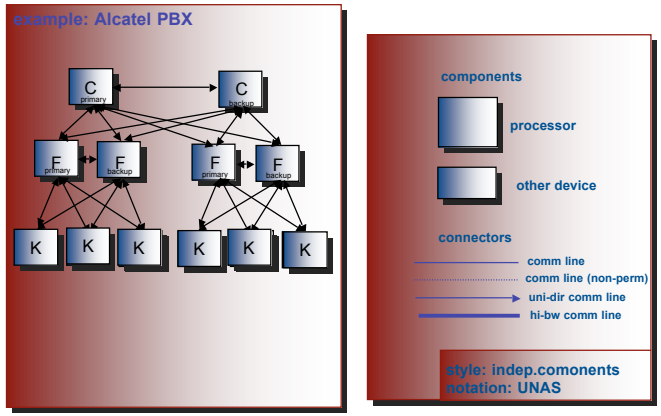





## The Physical Architecture

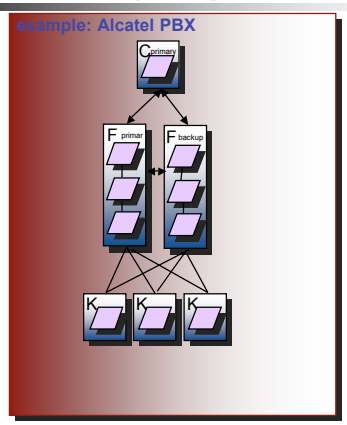
- Represented by Physical View
  - of interest to system designer
  - concerns: scalability, performance, availability, reliability
  - presents how processes, objects etc. are mapped onto processing nodes
- Components:processing nodes
- Connectors: LAN, WAN, bus,…
- Containers: Physical Subsystem
 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

## example: physical view






## Physical view (with process allocation)



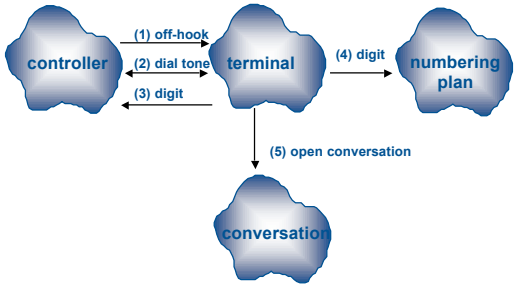


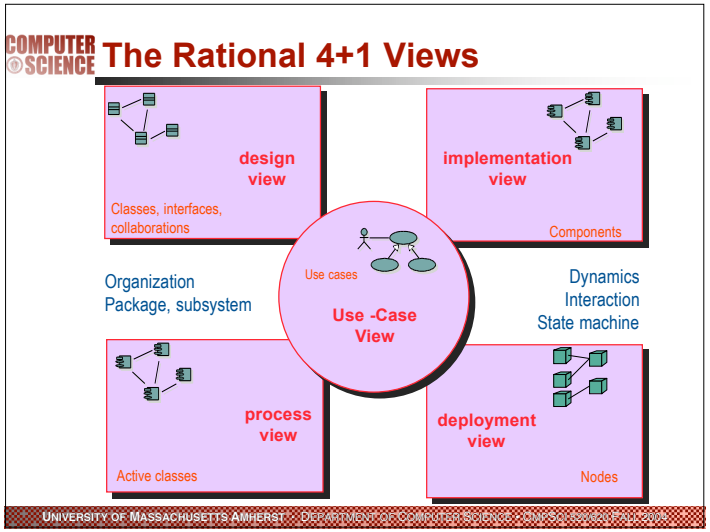

## Scenarios

- 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




COMPUTER SCIENCE
The Rational 4+1 Views
design view
Classes, interfaces, collaborations
implementation view
Components
Use cases
Use -Case View
Organization
Package, subsystem
Dynamics
Interaction
State machine
process view
Active classes
deployment view
Nodes
UNIVERSITY OF MASSACHUSETTS AMHERST