An Overview of Software Design Principles and Concepts for Communication Systems

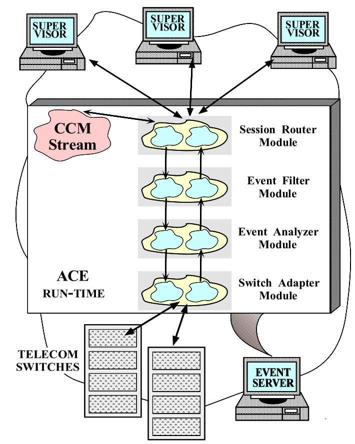
Douglas C. Schmidt

schmidt@cs.wustl.edu

Washington University, St. Louis
www.cs.wustl.edu/∼schmidt/

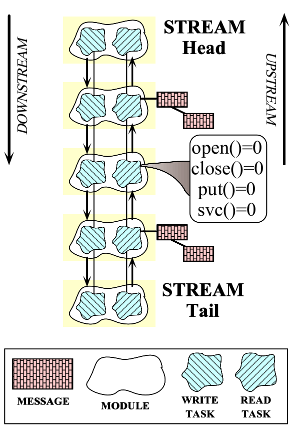
Sponsors
NSF, DARPA, BBN, Bellcore, Boeing, CDI/GDIS, Kodak, Lockheed, Lucent, Microsoft, Motorola, Nokia, Nortel, OTI, SAIC, Siemens SCR, Siemens MED, Siemens ZT, Sprint, USENIX

Goals of the Design Phase



- Decompose system into components
  - *i.e.*, identify the software architecture
- Determine relationships between components
  - *e.g.*, identify component dependencies and determine intercomponent communication mechanisms

Goals of the Design Phase (cont'd)



- Specify component interfaces
  - Interfaces should be well-defined
    * Facilitates component testing and team communication
- Describe component functionality
  - *e.g.*, informally or formally
- Identify opportunities for systematic reuse
  - Both top-down and bottom-up

Macro Steps in the Design Process

- In the design process the orientation moves from
  - Customer to developer
  - *What* to *how*
- Macro steps include:

1. *Preliminary Design*
   - External design describes the real-world model
   - Architectural design decomposes the requirement specification into software subsystems
2. *Detailed Design*
   - Specify each subsystem
   - Further decomposed subsystems, if necessary

Micro Steps in the Design Process

- Design is an iterative decision process with the following general steps:
  1. List the difficult decisions and decisions likely to change
  2. Design a component specification to hide each such decision
     - Make decisions that apply to whole program family first
     - Modularize *most likely* changes first
     - Then modularize remaining difficult decisions and decisions likely to change
     - Design the *uses* hierarchy as you do this (include reuse decisions)
  3. Treat each higher-level component as a specification and apply above process to each
  4. Continue refining until all design decisions are:
     - hidden in a component
     - contain easily comprehensible components
     - provide individual, independent, low-level implementation assignments

Key Design Concepts and Principles

Key design concepts and design principles include:

- *Decomposition*
- *Abstraction*
- *Information Hiding*
- *Modularity*
- *Extensibility*
- *Virtual Machine Structuring*
- *Hierarchy*
- *Program Families and Subsets*

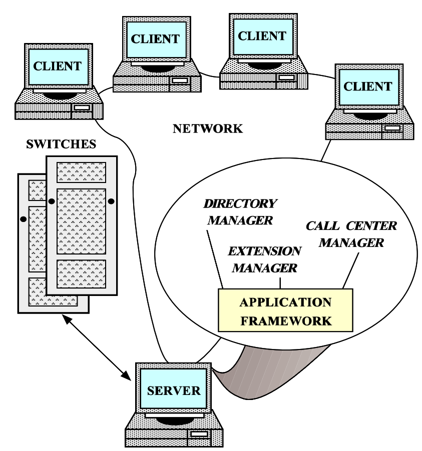
Main goal of these concepts and principles is to:

- Manage software system complexity
- Improve software quality factors
- Faciliate systematic reuse

Decomposition

- Motivation: handle complexity by spliting large problems into smaller problems, *i.e.*, "divide and conquer"
- Basic methodology:
  1. Select a piece of the problem (initially, the whole problem)
  2. Determine the components in this piece using a design paradigm, *e.g.*, functional, structured, object-oriented, generic, etc.
  3. Describe the components interactions
  4. Repeat steps 1 through 3 until some termination criteria is met
     - *e.g.*, customer is satisfied, run out of money, etc. ;-)

Decomposition Example: External OS for PBX



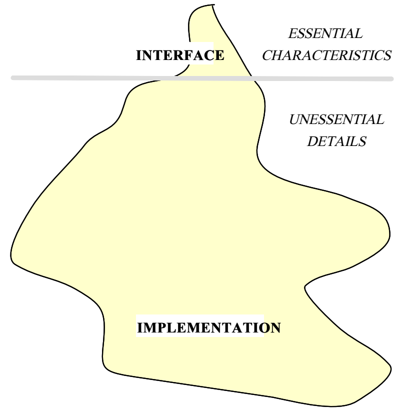
- **Features**
  - Allow clients to manage various aspects of PBX switches without modifying the switch software
  - Support reuse of existing components based on a common architectural framework

www.cs.wustl.edu/~schmidt/DSEJ-94.ps.gz

Decomposition Principles

1. Don't design components to correspond to execution steps
   - Since design decisions usually transcend execution time
2. Decompose so as to limit the effect of any one design decision on the rest of the system
   - Anything that permeates the system will be expensive to change
3. Components should be specified by all information needed to use the component
   - and *nothing more*!

Abstraction

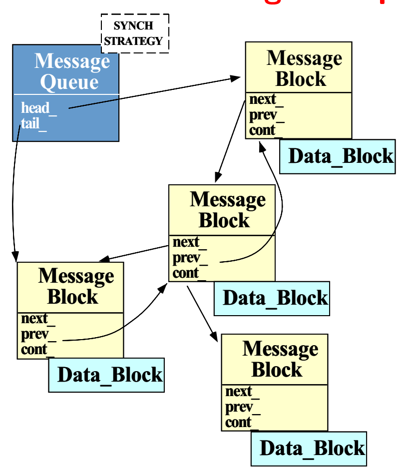


- Motivation: manage complexity by emphasizing *essential characteristics* and suppressing *implementation details*
- Common abstractions
  1. Procedural abstraction
     – *e.g.*, closed subroutines
  2. Data abstraction
     – *e.g.*, ADTs
  3. Control abstraction
     – *e.g.*, iterators, loops, and multitasking

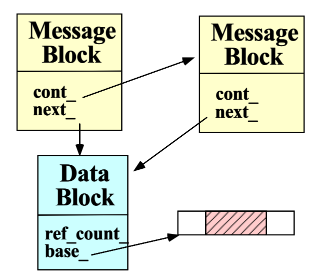
Information Hiding

- Motivation: design decisions that are subject to change should be hidden behind abstract interfaces
  – *i.e.*, components
- Components should communicate only through well-defined interfaces.
- Each component is specified by as little information as possible.
- If internal details change, client components should be minimally affected
  – May not even require recompilation and relinking...
- Information hiding is one means to enhance abstraction

Information Hiding Example: ACE Message Queueing



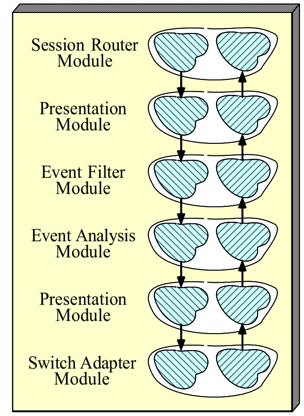
- `Message_Queue` and `Message_Block` hide Stream messaging implementations from clients
- *e.g.*, reference counting can be added transparently



Typical Information to be Hidden

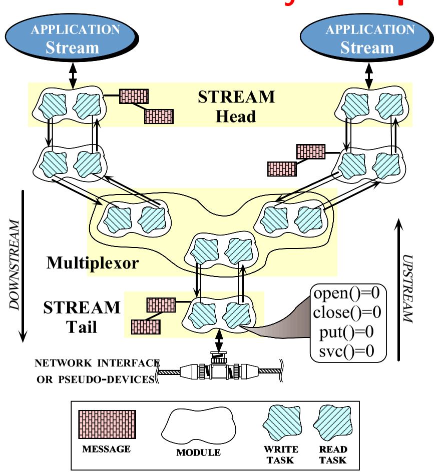
- Data representations
  - *i.e.*, using abstract data types
- Algorithms
  - *e.g.*, sorting or searching techniques
- Input and Output Formats
  - Machine dependencies, *e.g.*, byte-ordering, character codes
- Policy/mechanism distinctions
  - *e.g.*, OS scheduling, garbage collection, process migration
- Lower-level component interfaces
  - *e.g.*, ordering of low-level operations, *i.e.*, process sequence

Modularity



- A *Modular system* is one that's structured into identifiable abstractions called *components*
  - Components should possess well-specified *abstract interfaces*
  - Components should have high *cohesion* and low *coupling*
- Modularity is important for both design and implementation phases

Modularity Example: ACE `Stream`



- A `Stream` contains a stack of `Modules`
- Each `Module` contains two `Tasks`
  - *i.e.*, a *read* `Task` and a *write* `Task`
- Each `Task` contains a `Message_Queue` and a pointer to a `Thread_Manager`

Component Definitions

- A component is
  - A software entity encapsulating the representation of an abstraction, *e.g.*, an ADT
  - A vehicle for hiding at least one design decision
  - A "work" assignment for a programmer or group of programmers
  - A unit of code that
    - * has one or more names
    - * has identifiable boundaries
    - * can be (re-)used by other components
    - * encapsulates data
    - * hides unnecessary details
    - * can be separately compiled (if supported)

Component Interfaces

- A component interface consists of several sections:
  - Imports
    - Services requested from other components
  - Exports
    - Services provided to other components
  - Access Control
    - *e.g.*, protected/private/public
- Heuristics for determining component interfaces:
  - Define one specification that allows multiple implementations
  - Anticipate change
    - *e.g.*, use objects for parameters

Benefits of Modularity

Modularity facilitates software quality factors, *e.g.*,:

- Extensibility → well-defined, abstract interfaces
- Reusability → low-coupling, high-cohesion
- Compatibility → design "bridging" interfaces
- Portability → hide machine dependencies

Modularity is important for good designs since it:

- Enhances for *separation of concerns*
- Enables developers to reduce overall system complexity via *decentralized* software architectures
- Increases *scalability* by supporting independent and concurrent development by multiple personnel

Criteria for Evaluating Modular Designs

Component decomposability

- Are larger components decomposed into smaller components?

Component composability

- Are larger components composed from existing smaller components?

Component understandability

- Are components separately understandable?

Component continuity

- Do small changes to the specification affect a localized and limited number of components?

Component protection

- Are the effects of run-time abnormalities confined to a small number of related components?

Principles for Ensuring Modular Designs

Language support for components

- Components should correspond to syntactic units in the language

Few interfaces

- Every component should communicate with as few others as possible

Small interfaces (weak coupling)

- If any two components communicate at all, they should exchange as little information as possible

Explicit Interfaces

- Whenever two components A and B communicate, this must be obvious from the text of A or B or both

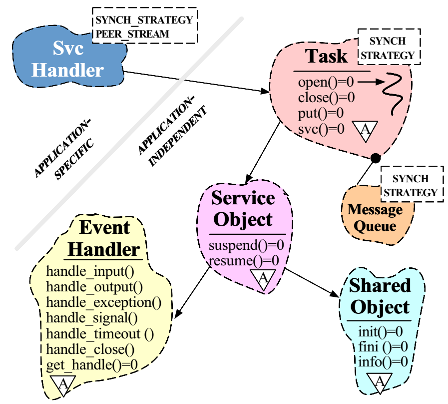
Information Hiding

- All information about a component should be private unless it's specifically declared public

Extensibility

- Motivation: aspects of a design "seem" constant until they are examined in the light of the dependency structure of an application
  - At this point, it becomes necessary to refactor the framework or pattern to account for the variation
- Therefore, components often must be *both* open and closed, *i.e.*, the "open/closed" principle:
  - Open component → still available for extension
    * This is necessary since the requirements and specifications are rarely completely understood from the system's inception
  - Closed component → available for use by other components
    * This is necessary since code sharing becomes unmanageable when reopening a component triggers many changes

Extensibility Example: ACE `Task`

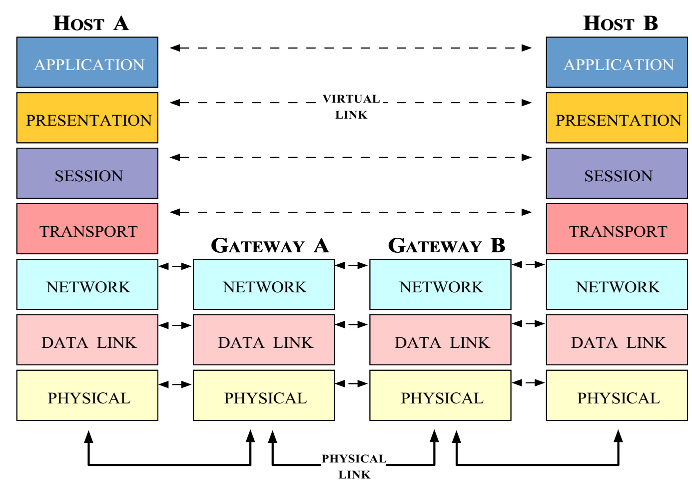


- **Features**
  - `Tasks` can register with a `Reactor`
  - They can be dynamically linked
  - They can queue data
  - They can run as "active objects"
- Note how OO techniques use inheritance and dynamic binding to produce components that are *both* open and closed

Virtual Machine Structuring

- Motivation: decompose system into smaller, more manageable units, that are *layered* hierarchically
- A virtual machine provides an extended "software instruction set"
  - Extensions provide additional data types and associated "software instructions"
  - Modeled after hardware instruction set primitives that work on a limited set of data types
- A virtual machine component provides a set of operations that are useful in developing a *family* of similar systems

Virtual Machine Example: OSI Protocol Stack



Other Examples of Virtual Machines

Computer architectures

- *e.g.*, compiler → assembler → obj code → microcode → gates, transistors, signals, etc.

Operating systems

- *e.g.*, Mach, BSD UNIX

Hardware Machine	Software Virtual Machine
instruction set	set of system calls
restartable instructions	restartable system calls
interrupts/traps	signals
interrupt/trap handlers	signal handlers
blocking interrupts	masking signals
interrupt stack	signal stack

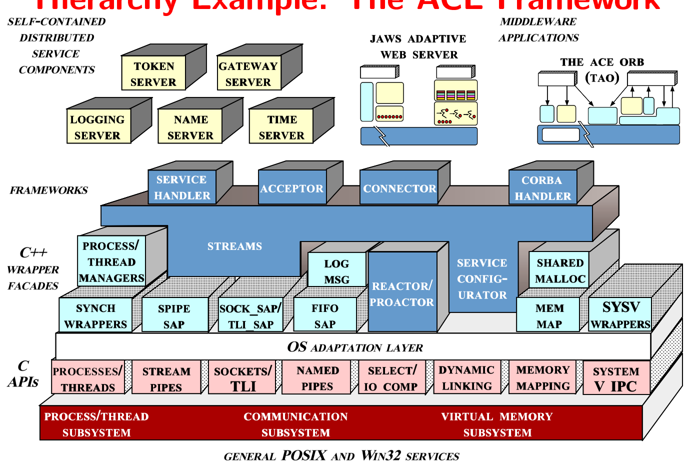
Java Virtual Machine (JVM)

- Abstracts away from details of the underlying OS

Hierarchy

- Motivation: reduces component interactions by restricting the topology of relationships
- A relation defines a hierarchy if it partitions units into levels (note connection to *virtual machines*)
  - Level 0 is the set of all units that use no other units
  - Level i is the set of all units that use at least one unit at level $< i$ and no unit at level $\geq i$.
- Hierarchies form the basis of *architectures* and *designs*
  - Facilitates independent development
  - Isolates ramifications of change
  - Allows rapid prototyping

Hierarchy Example: The ACE Framework



www.cs.wustl.edu/∼schmidt/ACE.html

Defining Hierarchies

- Relations that define hierarchies:
  - *Uses*
  - *Is-Composed-Of*
  - *Is-A*
  - *Has-A*
- The first two are general to all design methods, the latter two are more particular to OO design and programming

The Uses Relation

- X Uses Y if the correct functioning of X depends on the availability of a correct implementation of Y
- Note, *uses* is not necessarily the same as *invokes*:
  - Some invocations are not uses
    - *e.g.*, error logging
  - Some uses don't involve invocations
    - *e.g.*, message passing, interrupts, shared memory access
- A *uses* relation does not necessarily yield a hierarchy (avoid cycles...)

The Uses Relation (cont'd)

- Allow X to use Y when:
  - X is simpler because it uses Y
    - *e.g.*, Standard C library routines
  - Y is not substantially more complex because it is not allowed to use X
    - *i.e.*, hierarchies should be semantically meaningful
  - there is a useful subset containing Y and not X
    - *i.e.*, allows sharing and reuse of Y
  - there is no conceivably useful subset containing X but not Y
    - *i.e.*, Y is necessary for X to function correctly

The Uses Relation

- How should recursion be handled?
  - Group X and Y as a single entity in the uses relation
- A hierarchy in the *uses* relation is essential for designing non-trivial reusable software systems
- Note that certain software systems require some form of controlled violation of a uses *hierarchy*
  - *e.g.*, asynchronous communication protocols, call-back schemes, signal handling, etc.
  - *Upcalls* are one way to control these non-hierarchical dependencies
- *Rule of thumb*:
  - Start with an invocation hierarchy and eliminate those invocations (*i.e.*, "calls") that are not uses relationships

The Is-Composed-Of Relation

- The *is-composed-of* relationship shows how the system is broken down in components
- X *is-composed-of* $\{x_i\}$ if X is a group of units x_i that share some common purpose
- The system structure graph description can be specified by the *is-composed-of* relation such that:
  - non-terminal are "virtual" code
  - terminals are the only units represented by "actual" code

The Is-Composed-Of Relation

- Many programming languages support the *is-composed-of* relation via some higher-level component or record structuring technique
- Note: the following are not equivalent:
  - level (virtual machine)
  - component (an entity that hides a secret)
  - a subprogram (a code unit)
- Components and levels need not be identical, as a component may have several components on several levels of a uses hierarchy

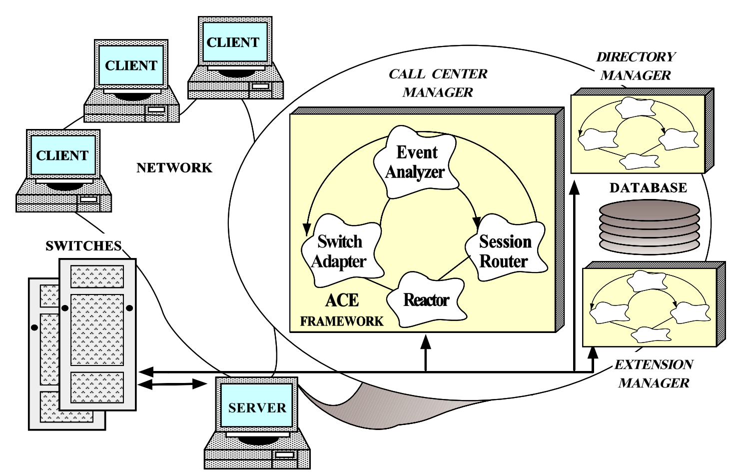
The Is-A and Has-A Relations

- These two relationships are associated with object-oriented design and programming languages that possess inheritance and classes.
- *Is-A* or *Descendant* relationship
  - class X possesses *Is-A* relationship with class Y if instances of class X are specialization of class Y.
  - *e.g.*, a square is a specialization of a rectangle, which is a specialization of a shape...
- *Has-A* or *client* relationship
  - class X possesses a *Has-B* relationship with class Y if instances of class X contain an instance(s) of class Y.
  - *e.g.*, a car has an engine and four tires...

Program Families and Subsets

- Motivation: facilitate *extension* and *contraction* of large-scale software systems
  - *e.g.*, the ACE framework
- Program families are natural way to detect and implement *subsets*
  - Minimize footprints for embedded systems
  - Promotes reusability
  - Anticipates potential changes
- Heuristics for identifying subsets:
  - Analyze requirements to identify minimally useful subsets
  - Also identify minimal increments to subsets

Example of Program Families: External OS for PBX



Other Examples of Program Families and Subsets

- Different services for different markets
  - *e.g.*, different alphabets, different vertical applications, different I/O formats
- Different hardware or software platforms
  - *e.g.*, compilers or OSs
- Different resource trade-offs
  - *e.g.*, speed vs space
- Different internal resources
  - *e.g.*, shared data structures and library routines
- Different external events
  - *e.g.*, UNIX I/O device interface
- Backward compatibility
  - *e.g.*, sometimes it is important to retain bugs!

Concluding Remarks

- Good designs generally can be boiled down to a few key principles:
  - Separate interface from implementation
  - Determine what is *common* and what is *variable* with an interface and an implementation
  - Allow substitution of *variable* implementations via a *common* interface
    - *i.e.*, the "open/closed" principle
  - Dividing *commonality* from *variability* should be goal-oriented rather than exhaustive
- Design is not simply the act of drawing a picture using a CASE tool or using graphical UML notation!!!
  - Design is a fundamentally *creative* activity