

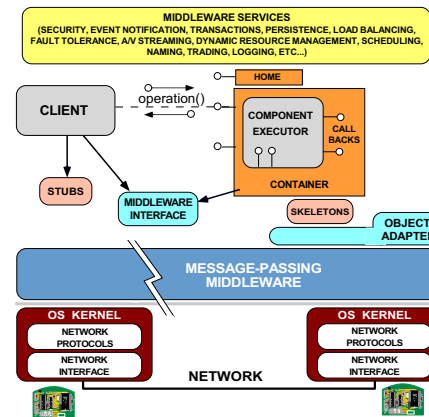
Software Design Principles and Guidelines

Douglas C. Schmidt

d.schmidt@vanderbilt.edu
 Vanderbilt University, St. Louis
 www.cs.wustl.edu/~schmidt/

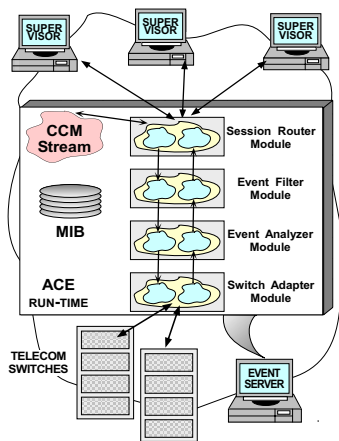
May 25, 2003

Design Principles and Guidelines Overview



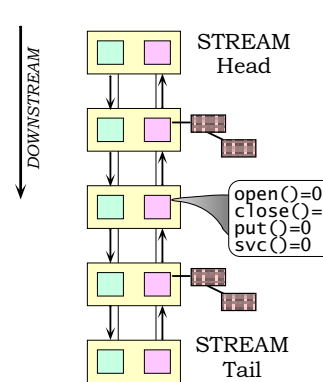
- Design Principles
 - Important design concepts
 - Useful design principles
- Development Methodologies
 - Traditional approaches
 - Agile programming
- Design Guidelines
 - Motivation
 - Common Design Mistakes
 - Design Rules

Motivation: Goals of the Design Phase (1/2)



- Decompose system into components
 - *i.e.*, identify the software architecture
- Determine relationships between components
 - *e.g.*, identify component dependencies
- Determine intercomponent communication mechanisms
 - *e.g.*, globals, function calls, shared memory, IPC/RPC

Motivation: Goals of the Design Phase (2/2)



- 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

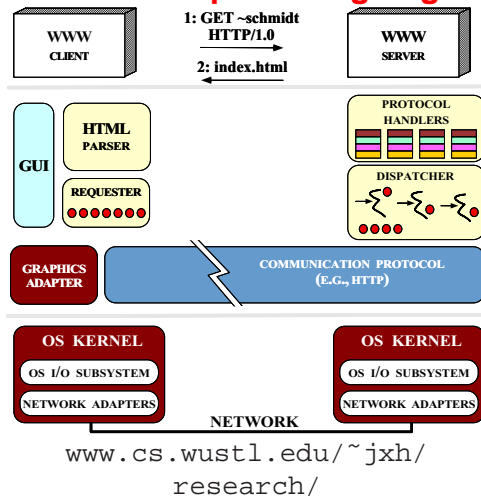
Macro Steps in the Design Process

- 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

- Given a requirements spec, *design* is an iterative decision process with the following general steps:
 1. List the hard 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

Example: Designing a Web Server



- **Web server design decisions**
 - Portability issues
 - I/O demuxing and concurrency
 - HTTP protocol processing
 - File access
- **Web server components**
 - Event dispatcher
 - Protocol handler
 - Cached virtual filesystem

Key Design Concepts and Principles

Key design concepts and design principles include:

1. *Decomposition*
2. *Abstraction and information hiding*
3. *Component modularity*
4. *Extensibility*
5. *Virtual machine architectures*
6. *Hierarchical relationships*
7. *Program families and subsets*

Main goal of these concepts and principles is to:

- Manage software system complexity
- Improve software quality factors
- Facilitate systematic reuse
- Resolve common design challenges

Challenge 1: Determining the Web Server Architecture

- **Context:** A large and complex production web server
- **Problems:**
 - Designing the web server as a large monolithic entity is tedious and error-prone
 - Web server developers must work concurrently to improve productivity
 - Portability and resuability are important quality factors

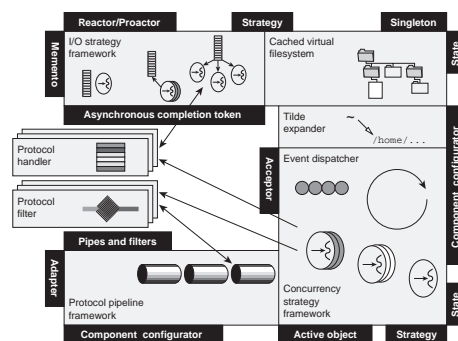
8

Solution: Decomposition

- Decomposition handles complexity by splitting large problems into smaller problems
- This “divide and conquer” concept is common to all life-cycle processes and design techniques
- 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 time/money, etc. ;-)

9

Decomposition Example: Web Server Framework



- **Features**
 - High-performance
 - Flexible concurrency, demuxing, and caching mechanisms
 - Uses frameworks based on ACE

www.cs.wustl.edu/~schmidt/PDF/JAWS.pdf

10

Object-Oriented Decomposition Principles

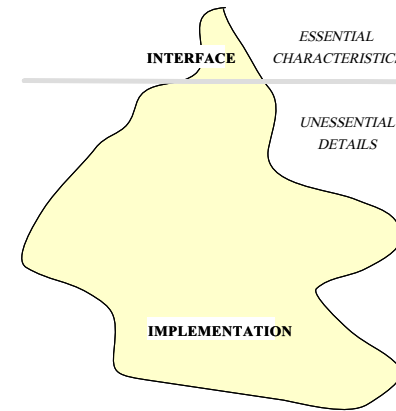
1. Don't design components to correspond to execution steps
 - Since design decisions usually transcend execution time
2. Decompose so as to limit the effect of any one design decision on the rest of the system
 - Anything that permeates the system will be expensive to change
3. Components should be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

11

Challenge 2: Implementing a Flexible Web Server

- **Context:** The requirements that a production web server must meet will change over time, e.g.:
 - New platforms
 - New compilers
 - New functionality
 - New performance goals
- **Problems:**
 - If the web server is “hard coded” using low-level system calls it will be hard to port
 - If web server developers write software that’s tightly coupled with internal implementation details the software will be hard to evolve

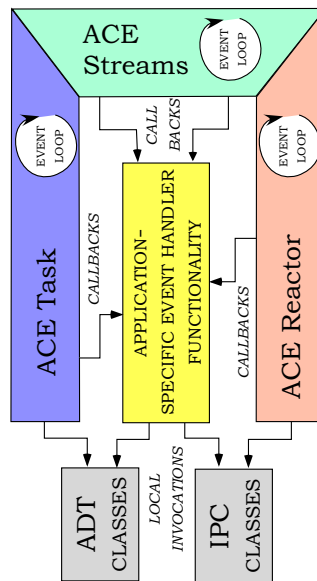
Solution: Abstraction



- Abstraction manages complexity by emphasizing *essential characteristics* and suppressing *implementation details*
- Allows postponement of certain design decisions that occur at various levels of analysis, e.g.,
 - Representational and algorithmic considerations
 - Architectural and structural considerations
 - External environment and platform considerations

Common Types of Abstraction

1. **Procedural abstraction**
 - e.g., closed subroutines
2. **Data abstraction**
 - e.g., ADT classes and component models
3. **Control abstraction**
 - e.g., loops, iterators, frameworks, and multitasking



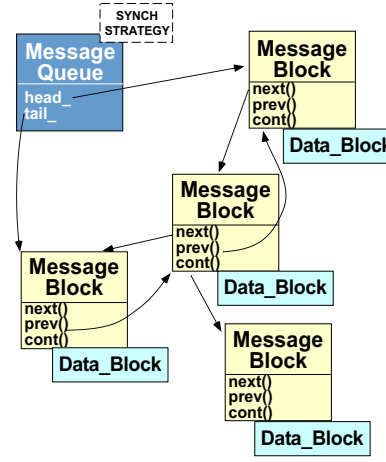
Information Hiding

- Information hiding is an important means of achieving abstraction
 - i.e., design decisions that are subject to change should be hidden behind abstract interfaces
- Application software should communicate only through well-defined interfaces
- Each interface should be specified by as little information as possible
- If internal details change, clients should be minimally affected
 - May not even require recompilation and relinking...

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
- **Lower-level interfaces**
 - *e.g.*, ordering of low-level operations, *i.e.*, process sequence
- **Separating policy and mechanism**
 - Multiple policies can be implemented by same mechanisms
 - * *e.g.*, OS scheduling and virtual memory paging
 - Same policy can be implemented by multiple mechanisms
 - * *e.g.*, reliable communication service can be provided by multiple protocols

Information Hiding Example: Message Queueing



- A Message_Queue is a list of ACE_Message_Blocks
- Efficiently handles arbitrarily-large message payloads
- Design encapsulates and parameterizes various aspects
 - *e.g.*, synchronization, memory allocators, and reference counting can be added transparently

The ACE_Message_Block Class

```

ACE_Message_Block
# rd_ptr_ : size_t
# wr_ptr_ : size_t
# cont_ : ACE_Message_Block *
# next_ : ACE_Message_Block *
# prev_ : ACE_Message_Block *
# data_block_ : ACE_Data_Block *

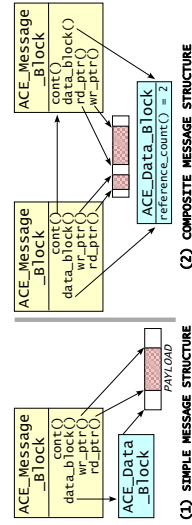
+ init (size : size_t) : int
+ msg_type (type : ACE_Message_Type)
+ msg_ptrloc (ptr : ACE_Message_Type)
+ msg_ptrloc (ptr : ACE_Message_Type)
+ clone () : ACE_Message_Block *
+ duplicate () : ACE_Message_Block *
+ release () : ACE_Message_Block *
+ set_flags (flags : u_long) : u_long
+ clr_flags (flags : u_long) : u_long
+ copy (buf : const char *, n : size_t) : int
+ rd_ptr (n : size_t)
+ wr_ptr (n : size_t)
+ rd_ptr (n : size_t)
+ wr_ptr (n : size_t)
+ rd_ptr () : char *
+ wr_ptr () : char *
+ total_length () : size_t
+ size () : size_t
    
```

```

classDiagram
    class ACE_Message_Block {
        rd_ptr_ : size_t
        wr_ptr_ : size_t
        cont_ : ACE_Message_Block *
        next_ : ACE_Message_Block *
        prev_ : ACE_Message_Block *
        data_block_ : ACE_Data_Block *
    }
    class ACE_Data_Block {
        base_ : char *
        refcnt_ : int
    }
    ACE_Message_Block <|-- ACE_Data_Block
    
```

Class characteristics

- Hide messaging implementations from clients



The ACE_Message_Queue Class

```

ACE_Message_Queue
# head_ : ACE_Message_Block *
# tail_ : ACE_Message_Block *
# high_water_mark_ : size_t
# low_water_mark_ : size_t

+ ACE_Message_Queue (high_water_mark : size_t = DEFAULT_HWM,
                    low_water_mark : size_t = DEFAULT_LWM,
                    notify : ACE_Notification_Strategy * = 0)
+ open (high_water_mark : size_t = DEFAULT_HWM,
        low_water_mark : size_t = DEFAULT_LWM,
        notify : ACE_Notification_Strategy * = 0) : int
+ flush () : int
+ notification_strategy (s : ACE_Notification_Strategy *) : void
+ is_empty () : int
+ get_tail () : ACE_Message_Block *
+ enqueue_tail (item : ACE_Message_Block *,
               timeout : ACE_Time_Value * = 0) : int
+ enqueue_head (item : ACE_Message_Block *,
               timeout : ACE_Time_Value * = 0) : int
+ enqueue_prio (item : ACE_Message_Block *,
               timeout : ACE_Time_Value * = 0) : int
+ dequeue_head (item : ACE_Message_Block *,
               timeout : ACE_Time_Value * = 0) : int
+ dequeue_tail (item : ACE_Message_Block *,
               timeout : ACE_Time_Value * = 0) : int
+ high_water_mark (new_hwm : size_t) : void
+ high_water_mark (void) : size_t
+ low_water_mark (new_lwm : size_t) : void
+ low_water_mark (void) : size_t
+ close () : int
+ activate () : int
+ pulse () : int
+ state () : int
    
```

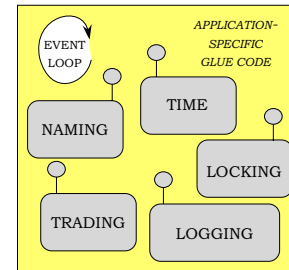
Class characteristics

- Note how the synchronization aspect can be strategized!

Challenge 3: Determining the Units of Web Server Decomposition

- **Context:** A production web server that uses abstraction and information hiding
- **Problems:**
 - Need to determine the appropriate units of decomposition, which should
 - * Possess well-specified *abstract interfaces* and
 - * Have high *cohesion* and low *coupling*

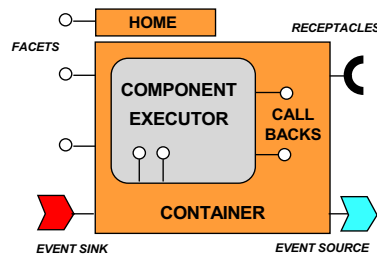
Solution: Component Modularity



- A *modular system* is one that's structured into identifiable abstractions called *components*
 - A software entity that represents an abstraction
 - A “work” assignment for developers
 - 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

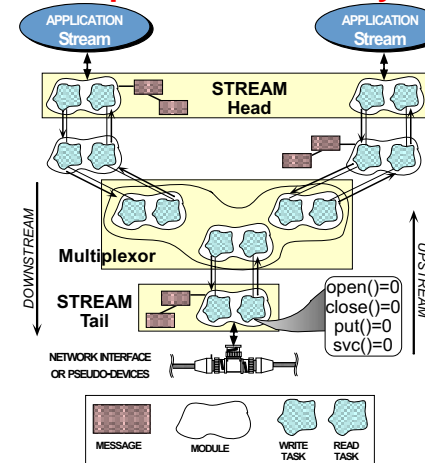
Designing Component Interfaces

- A component interface consists of several types of ports:
 - **Exports**
 - * Services provided to other components, *e.g.*, facets and event sources
 - **Imports**
 - * Services requested from other components, *e.g.*, receptacles and event sinks
 - **Access Control**
 - * Not all clients are equal, *e.g.*, protected/private/public



- Define components that provide multiple interfaces and implementations
- Anticipate change

Component Modularity Example: Stream Processing



- A Stream allows flexible configuration of layered processing modules
- A Stream component contains a stack of Module components
- Each Module contains two Task components
 - *i.e.*, *read* and *write* Tasks
- Each Task contains a Message_Queue component and a Thread_Manager component

Benefits of Component 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

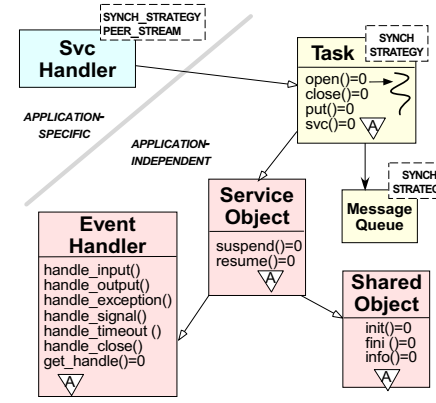
Challenge 4: “Future Proofing” the Web Server

- **Context:** A production web server whose requirements will change over time
- **Problems:**
 - Certain design aspects seem constant until they are examined in the overall structure of an application
 - Developers must be able to easily refactor the web server to account for new sources of variation

Solution: Extensibility

- Extensible software is important to support successions of quick updates and additions to address new requirements and take advantage of emerging opportunities/markets
- Extensible components 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: Active Object Tasks



• Features

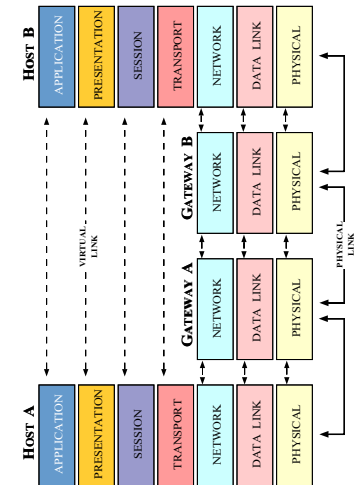
- Tasks can register with a Reactor
- They can be dynamically linked
- They can queue data
- They can run as “active objects”
- JAWS uses inheritance and dynamic binding to produce task components that are *both* open and closed

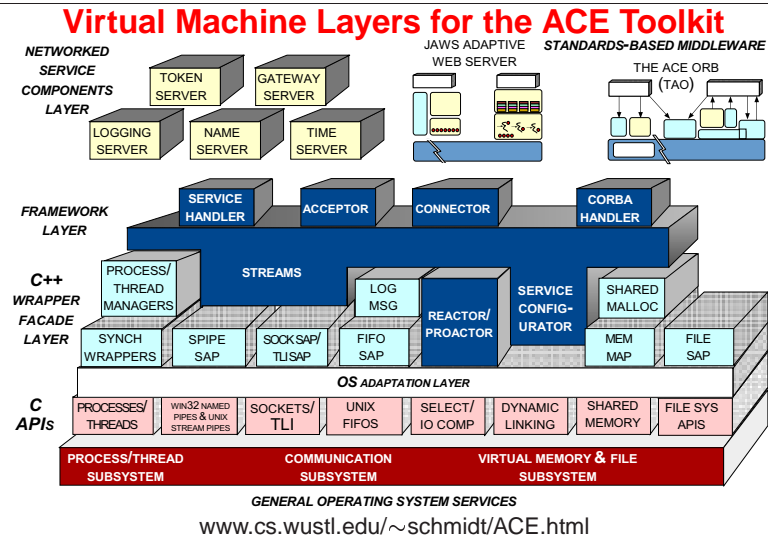
Challenge 5: Separating Concerns for Layered Systems

- **Context:** A production web server whose requirements will change over time
- **Problems:**
 - To enhance reuse and flexibility, it is often necessary to decompose a web server into smaller, more manageable units that are *layered* in order to
 - * Enhance reuse, *e.g.*, multiple higher-layer services can share lower-layer services
 - * Transparently and incrementally enhancement functionality
 - * Improve performance by allowing the selective omission of unnecessary service functionality
 - * Improve implementations, testing, and maintenance

Solution: Virtual Machine Architectures

- 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 layer provides a set of operations that are useful in developing a *family* of similar systems





Other Examples of Virtual Machines

Computer architectures

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

Operating systems

- e.g., Linux

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

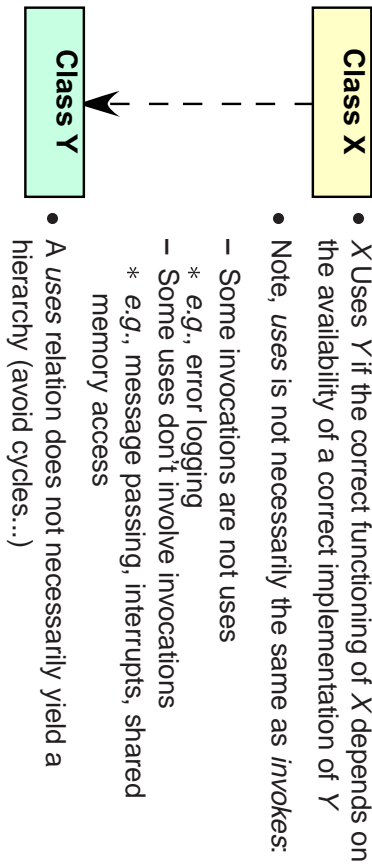
Challenge 6: Separating Concerns for Hierarchical Systems

- **Context:** A production web server whose requirements will change over time
- **Problems:**
 - Developers need to program components at different levels of abstraction independently
 - Changes to one set of components should be isolated as much as possible from other components
 - Need to be able to “visualize” the structure of the web server design

Solution: Hierarchical Relationships

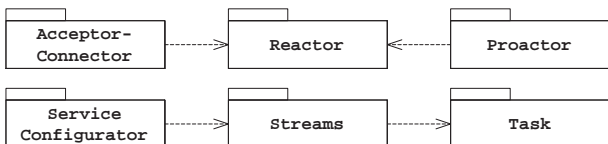
- Hierarchies reduce component interactions by restricting the topology of relationships
- A relation defines a hierarchy if it partitions units into levels (note connection to *virtual machine architectures*)
 - Level 0 is the set of all units that use no other units
 - Level i is the set of all units 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

The Uses Relation (1/3)

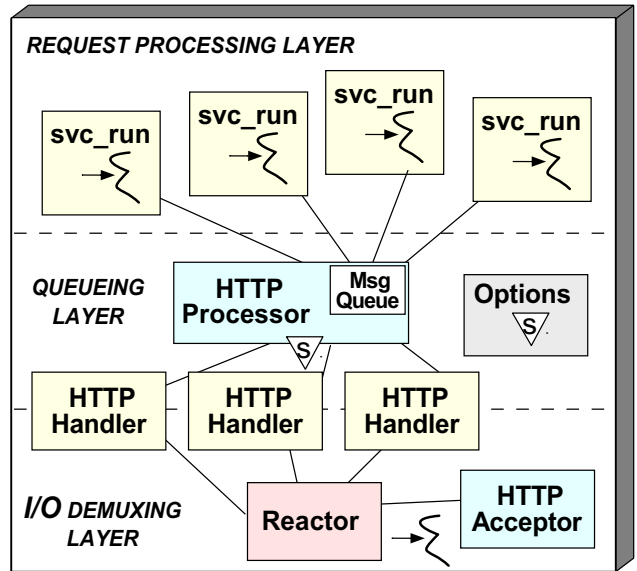


The Uses Relation (2/3)

- Allow X to use Y when:
 - X is simpler because it uses Y
 - * e.g., Standard C++ library classes
 - Y is not substantially more complex because it is not allowed to use X
 - 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
- Uses relationships can exist between classes, frameworks, subsystems, etc.

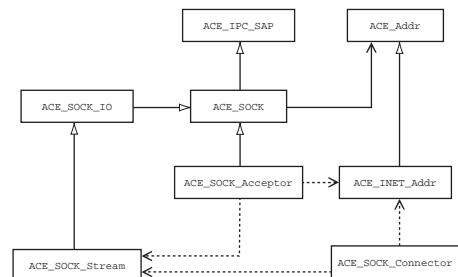


Hierarchy Example: JAWS Architecture



Defining Hierarchies

- Relations that define hierarchies include:
 - *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 (3/3)

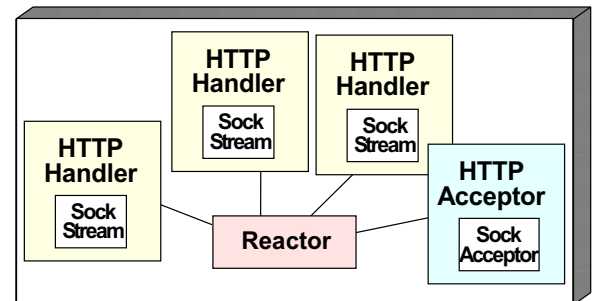
- A hierarchy in the *uses* relation is essential for designing reusable software systems
- However, certain software systems require controlled violation of a *uses hierarchy*
 - e.g., asynchronous communication protocols, OO callbacks in frameworks, signal handling, etc.
 - *Upcalls* are one way to 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

- Many programming languages support the *is-composed-of* relation via some higher-level component or record structuring technique
- However, the following are not equivalent:
 - level (virtual machine)
 - component (an entity that hides one or more “secrets”)
 - a subprogram (a code unit)
- Components and levels need not be identical, as a component may appear in several levels of a *uses hierarchy*

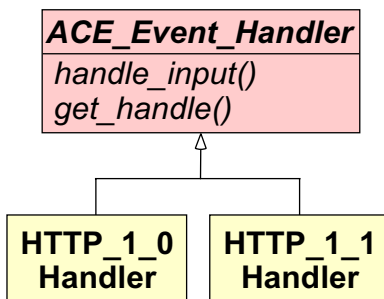
The Is-Composed-Of Relation

- 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 components x_i that share some common purpose
- The following diagram illustrates some of the *is-composed-of* relationships in JAWS



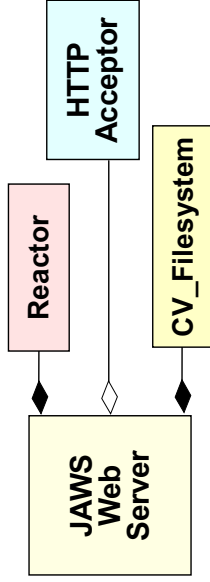
The Is-A Relation

- This “ancestor/descendant” relationship is associated with object-oriented design and programming languages that possess inheritance and dynamic binding
- class X possesses *Is-A* relationship with class Y if instances of class X are specialization of class Y.
 - e.g., an `HTTP_1_0_Handler` *Is-A* `ACE_Event_Handler` that is specialized for processing HTTP 1.0 requests



The Has-A Relation

- This “client” relationship is associated with object-oriented design and programming languages that possess classes and objects
- class X possesses a *Has-A* relationship with class Y if instances of class X contain an instance(s) of class Y.
- e.g., the JAWS web server *Has-A* *Reactor*, *HTTP_Acceptor*, and *CV_FileSystem*



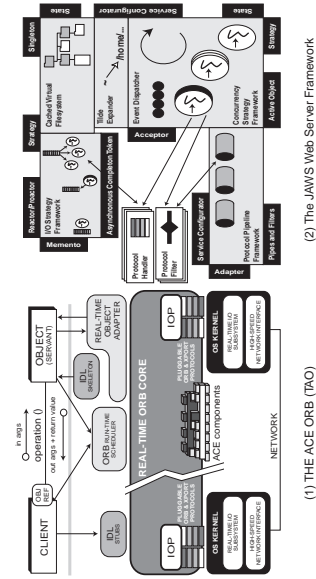
Challenge 7: Enabling Expansion and Contraction of Software

- **Context:** A production web server whose requirements will change over time
- **Problems:**
 - It may be necessary to reduce the overall functionality of the server to run in resource-constrained environments
 - To meet externally imposed schedules, it may be necessary to release the server without all the features enabled

Solution: Program Families and Subsets

- This principle should be applied to facilitate *extension* and *contraction* of large-scale software systems, particularly reusable middleware infrastructure
 - e.g., JAWS, ACE, etc.
- Program families are natural way to detect and implement *subsets*
 - Minimize footprints for embedded systems
 - Promotes system 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: JAWS and TAO



- TAO is a high-performance, real-time implementation of the CORBA specification
- JAWS is a high-performance, adaptive Web server that implements the HTTP specification
- JAWS and TAO were developed using the wrapper facades and frameworks provided by the ACE toolkit

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!

48

Conventional Development Processes

- Waterfall Model
 - Specify, analyze, implement, test (**in sequence**)
 - Assumes that requirements can be specified up front
- Spiral Model
 - Supports iterative development
 - Attempts to assess risks of changes
- Rapid Application Development
 - Build a prototype
 - Ship it :-)

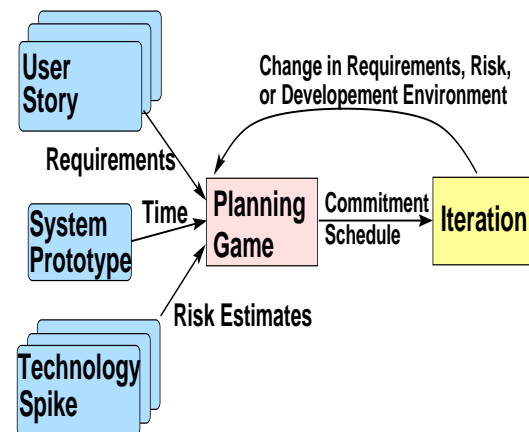
49

Agile Processes

- Stresses customer satisfaction, and therefore, involvement
 - Provide what the customer wants, as quickly as possible
 - Provide *only* what the customer wants
- Encourages changes in requirements
- Relies on testing
- For example, **eXtreme Programming** practices
 - Planning, designing, coding, testing

50

eXtreme Programming: Planning



based on <http://www.extremeprogramming.org/rules/planninggame.html>

- Start with *user stories*
 - Written by customers, to specify system requirements
 - Minimal detail, typically just a few sentences on a card
 - Expected development time: 1 to 3 weeks each, roughly
- Planning game creates commitment schedule for entire project
- Each iteration should take 2-3 weeks

51

eXtreme Programming: Designing

- Defer design decisions as long as possible
- Advantages:
 - Simplifies current task (just build what is needed)
 - You don't need to maintain what you haven't built
 - Time is on your side: you're likely to learn something useful by the time you need to decide
 - Tomorrow may never come: if a feature isn't needed now, it might never be needed
- Disadvantages:
 - Future design decisions may require rework of existing implementation
 - Ramp-up time will probably be longer later
 - * Therefore, always try to keep designs as simple as possible

eXtreme Programming: Coding

- *Pair programming*
 - *Always* code with a partner
 - *Always* test as you code
- Pair programming pays off by supporting good implementation, reducing mistakes, and exposing more than one programmer to the design/implementation
- If any deficiencies in existing implementation are noticed, either fix them or note that they need to be fixed

eXtreme Programming: Testing

- Unit tests are written *before* code
- Code **must** pass both its unit test **and** all regression tests before committing
- In effect, the test suite defines the system requirements
 - Significant difference from other development approaches
 - If a bug is found, a test for it **must** be added
 - If a feature isn't tested, it can be removed

Agile Processes: Information Sources

- Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, ISBN 0201616416, 1999
- Kent Beck, "Extreme Programming", *C++ Report* 11:5, May 1999, pp. 26–29+
- John Vlissides, "XP", interview with Kent Beck in the Pattern Hatching Column, *C++ Report* 11:6, June 1999, pp. 44-52+
- Kent Beck, "Embracing Change with Extreme Programming", *IEEE Computer* 32:10, October 1999, pp. 70-77
- <http://www.extremeprogramming.org/>
- <http://www.xprogramming.com/>
- <http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap>

Design Guidelines: Motivation

- Design is the process of organizing structured solutions to tasks from a problem domain
- This process is carried out in many disciplines, in many ways
 - There are many similarities and commonalities among design processes
 - There are also many common design mistakes . . .
- The following pages provide a number of “design rules.”
 - Remember, these rules are simply suggestions on how to better organize your design process, *not* a recipe for success!

Common Design Mistakes (1/2)

- Depth-first design
 - only partially satisfy the requirements
 - experience is best cure for this problem . . .
- Directly refining requirements specification
 - leads to overly constrained, inefficient designs
- Failure to consider potential changes
 - always design for extension and contraction
- Making the design too detailed
 - this overconstrains the implementation

Common Design Mistakes (2/2)

- Ambiguously stated design
 - misinterpreted at implementation
- Undocumented design decisions
 - designers become essential to implementation
- Inconsistent design
 - results in a non-integratable system, because separately developed modules don't fit together

Rules of Design (1/8)

- *Make sure that the problem is well-defined*
 - All design criteria, requirements, and constraints, should be enumerated before a design is started
 - This may require a “spiral model” approach
- *What comes before how*
 - *i.e.*, define the service to be performed at every level of abstraction before deciding which structures should be used to realize the services
- *Separate orthogonal concerns*
 - Do not connect what is independent
 - Important at many levels and phases . . .

Rules of Design (2/8)

- *Design external functionality before internal functionality.*
 - First consider the solution as a black-box and decide how it should interact with its environment
 - Then decide how the black-box can be internally organized. Likely it consists of smaller black-boxes that can be refined in a similar fashion
- *Keep it simple.*
 - Fancy designs are buggier than simple ones; they are harder to implement, harder to verify, and often less efficient
 - Problems that appear complex are often just simple problems huddled together
 - Our job as designers is to identify the simpler problems, separate them, and then solve them individually

Rules of Design (3/8)

- *Work at multiple levels of abstraction*
 - Good designers must be able to move between various levels of abstraction quickly and easily
- *Design for extensibility*
 - A good design is “open-ended,” *i.e.*, easily extendible
 - A good design solves a class of problems rather than a single instance
 - Do not introduce what is immaterial
 - Do not restrict what is irrelevant
- *Use rapid prototyping when applicable*
 - Before implementing a design, build a high-level prototype and verify that the design criteria are met

Rules of Design (4/8)

- Details should depend upon abstractions
 - Abstractions should not depend upon details
 - Principle of Dependency Inversion
- The granule of reuse is the same as the granule of release
 - Only components that are released through a tracking system can be effectively reused
- Classes within a released component should share common closure
 - That is, if one needs to be changed, they all are likely to need to be changed
 - *i.e.*, what affects one, affects all

Rules of Design (5/8)

- Classes within a released component should be reused together
 - That is, it is impossible to separate the components from each other in order to reuse less than the total
- The dependency structure for released components must be a DAG
 - There can be no cycles
- Dependencies between released components must run in the direction of stability
 - The dependee must be more stable than the dependor
- The more stable a released component is, the more it must consist of abstract classes
 - A completely stable component should consist of nothing but abstract classes

Rules of Design (6/8)

- Where possible, use proven patterns to solve design problems
- When crossing between two different paradigms, build an interface layer that separates the two
 - Don't pollute one side with the paradigm of the other

Rules of Design (7/8)

- Software entities (classes, modules, etc) should be open for extension, but closed for modification
 - The Open/Closed principle – Bertrand Meyer
- Derived classes must usable through the base class interface without the need for the user to know the difference
 - The Liskov Substitution Principle

Rules of Design (8/8)

- *Make it work correctly, then make it work fast*
 - Implement the design, measure its performance, and if necessary, optimize it
- *Maintain consistency between representations*
 - *e.g.*, check that the final optimized implementation is equivalent to the high-level design that was verified
 - Also important for documentation . . .
- Don't skip the preceding rules!
 - Clearly, this is the most frequently violated rule!!! ;-)

Concluding Remarks

- Good designs can generally be distilled into 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