

Software Engineering

G22.2440-001

Session 6 - Main Theme

From Analysis and Design to Software Architectures

Dr. Jean-Claude Franchitti

*New York University
Computer Science Department
Courant Institute of Mathematical Sciences*

1

Agenda

- Towards Agile Enterprise Architecture Models
- Building an Object Model Using UML
- Architectural Analysis
- Design Patterns
- Architectural Patterns
- Architecture Design Methodology
 - Achieving Optimum-Quality Results
 - Selecting Kits and Frameworks
 - Using Open Source vs. Commercial Infrastructures
- Summary
 - Individual Assignment #4
 - Project (Part 2)

2

Summary of Previous Session

- Traditional Data and Process Modeling Approaches
- From Requirements Analysis to Business and Application Models
- Roles of Software Analysis and Design
- Object-Oriented Analysis and Design with UML
- Selecting and Combining Approaches
- Creating a Data Model
- Homework #3
- Project #1 (ongoing)
- Summary

3

Part I

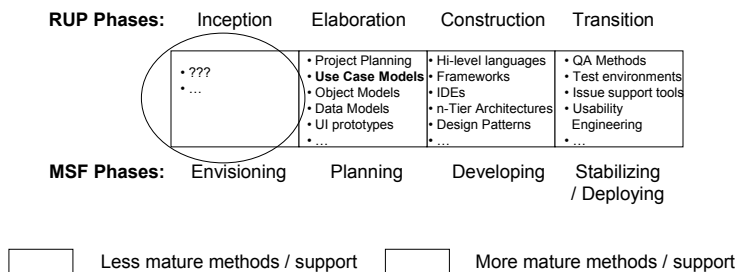
Towards Agile Enterprise Architecture Models

4

There is an Urgent Need for More Reliable, Concept-driven, Early Requirements Processes

(review)

- SDLC Process model frameworks have matured and converged significantly on effective practices in system development phases, however, they are still *weakest on the front-end*.



5

XP Practices

(review)

- The Planning Game** – determine the scope of the next release, based on business priorities and technical estimates.
- Small Releases** – every release should be as small as possible, containing the most valuable business requirements.
- Metaphor** – guide all development with a simple, shared story of how the whole system works.
- Simple Design** – design for the current functionality, not future needs, and make the design as simple as possible for that functionality (YAGNI).
- Refactoring** – ongoing redesign of software to improve its responsiveness to change.
- Testing** – developers write the unit tests before the code; unit tests are run “continuously”; customers write the functional tests.

6

XP Practices

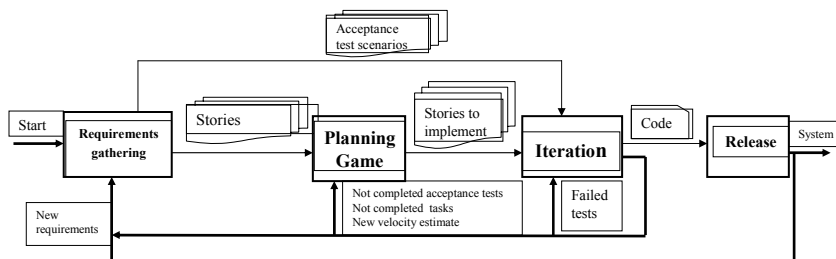
(review)

- ***Pair Programming*** – an extreme form of inspections; two people sit in front of the same terminal to write test cases and code.
- ***Collective Ownership*** – anyone can change any code anywhere in the system at any time.
- ***Continuous Integration*** – integrate and build the system many times a day, every time a task is finished.
- ***40-hour Week*** – work no more than 40 hours as a rule; never work overtime two weeks in a row.
- ***On-site Customer*** – real, live user on the team full time.
- ***Coding Standards*** – must be adhered to rigorously because all communication is through the code.

7

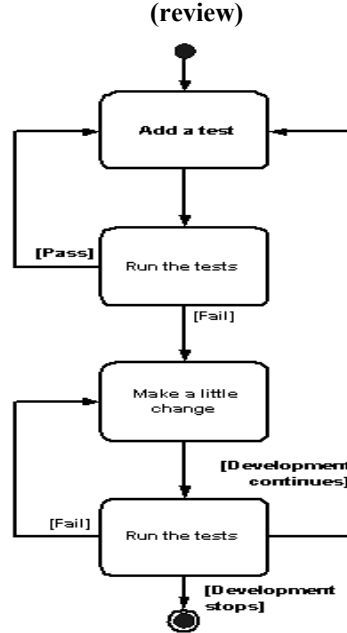
Overall XP Process

(review)



8

Steps of Test –Driven Development (TDD)



9

Do Agile Methods and Approaches Fill the Gap?

(review)

- Agile approaches recognize that continuous, effective communication among all stakeholders is absolutely central
 - they stress developing an attitude of mutual collaboration and invention
 - represent a significant step forward in the maturity and “realism” of software development methods
- But, Agile approaches also don’t offer mature, reliable techniques or assets with which to ensure that this type communication and joint decision making is fostered.
- Bottom line: NO
 - approaches of all kinds recognize the need to address the “envisioning” phase
 - there is definitely room for improvement for reliably facilitating envisioning

10

Emerging Technologies and Component Solutions are Revolutionizing ‘Business Capabilities’

(review)

- System development is no longer “green-field” development, but is now often more about:
 - configuring and integrating components,
 - innovatively leverage pre-built solution capabilities
 - creatively using new technologies.
- Methods are needed to help companies move from:
 - strategy to capability design
 - technology assessment to selection
 - Technology selection to solution roadmaps
- Tools are important
 - for decision making with consensus building, solution envisioning & design, strategic business case to project planning and project execution

11

Seeking a Repeatable Means to Mediate Strategic Communication and Requirements

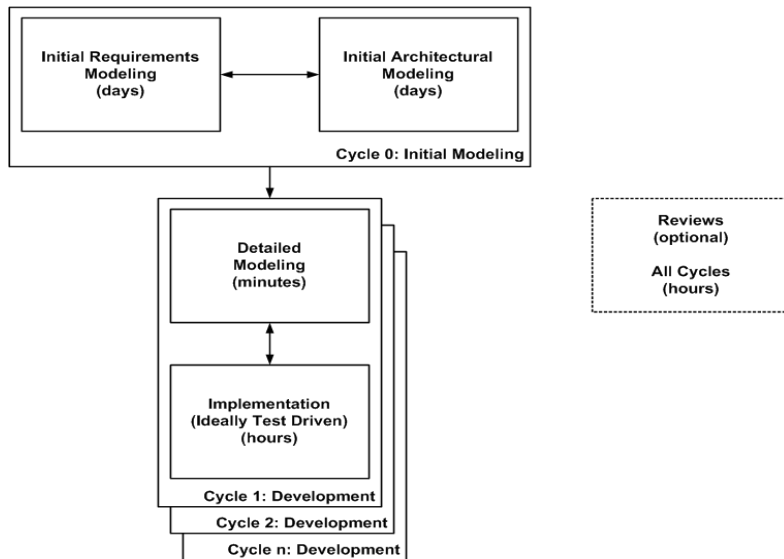
(review)

Candidate Approaches (illustrative, partial list):

- Business Service Patterns
- Business Centric Methodology
- Business Capabilities Design
- Capability Cases (note, members of the organizing committee will present an approach based on these)
- Agile Enterprise Architecture Models
- ...

12

Agile Model-Driven Development Lifecycle (review)



13

Part II

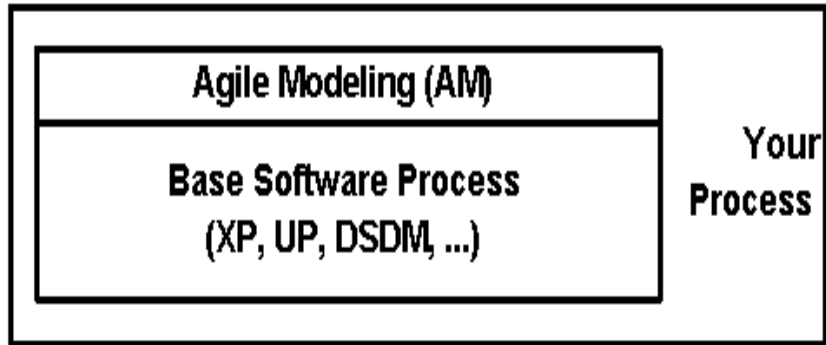
Building an Object Model Using UML: Agile Analysis and Design

See: <http://www.rspa.com/reflib/AgileDevelopment.html>

14

AM Enhances Other Software Processes

(review)



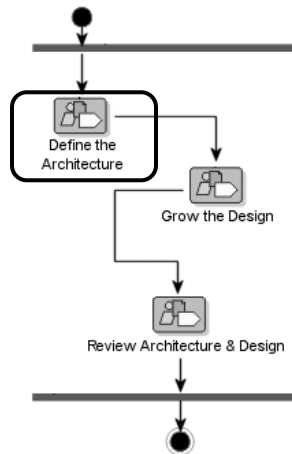
15

Part III

Architectural Analysis ala RUP

16

Define the Architecture



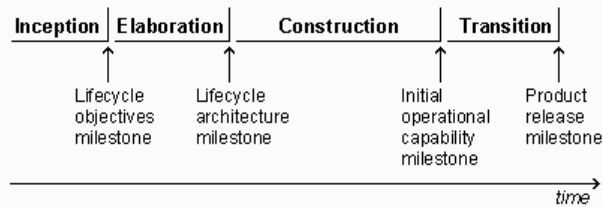
17

Some Architecture Concepts

- Packages
- Patterns and frameworks
- Architecture styles
- Layered architecture style
- Architecture Mechanisms

18

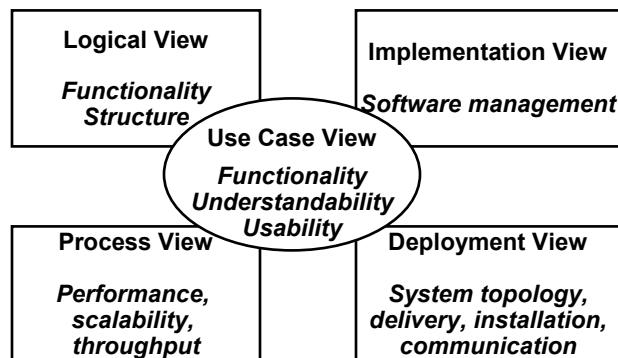
Architecture is the Focus of the RUP Elaboration Phase



- Elaboration Phase Goal: Baseline the architecture of the system
 - Provide a stable basis for the bulk of the design and implementation effort in the construction phase
 - The architecture evolves out of a consideration of the most significant requirements and an assessment of risk

19

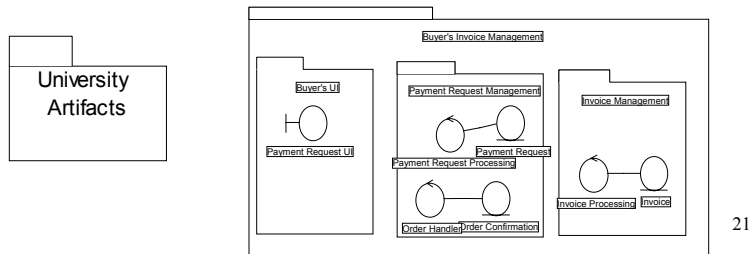
Focusing on the Logical View



20

Package

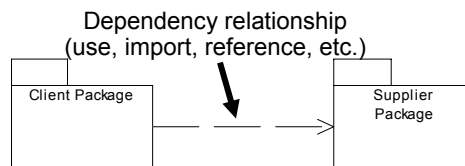
- A package is a general-purpose mechanism for organizing elements into groups
 - A model element that can contain other model elements
 - Typically group related, cohesive elements
- Use a package to
 - Organize the model under development
 - Provide a unit of configuration management



21

Package Dependency Relationships

- Packages can be related to one another using a dependency relationship
 - This defines the allowable relationships between the contained model elements



- Dependency implies
 - Changes to the supplier package may affect the client package (may need to recompile and re-test)
 - The client package cannot be reused independently because it depends on the supplier package

22

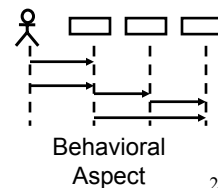
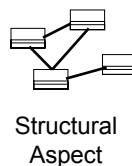
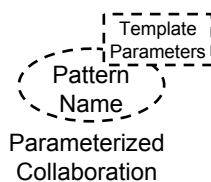
Patterns and Frameworks

- Pattern
 - A common solution to a common problem in a context
- Analysis/Design Pattern
 - A solution to a narrowly-scoped technical problem
 - A fragment of a solution, a partial solution, or a piece of the puzzle
- Framework
 - Defines the general approach to solving the problem
 - Skeletal solution, whose details may be analysis/design patterns

23

Design Patterns

- A design pattern is a solution to a common design problem
 - Describes a common design problem
 - Describes a proven solution to the problem
 - A solution based on experience
 - Discusses the result and trade-offs of applying the pattern
- Design patterns provide the capability to reuse successful designs
- A pattern is modeled as a parameterized collaboration in UML, including its structural and behavioral aspects



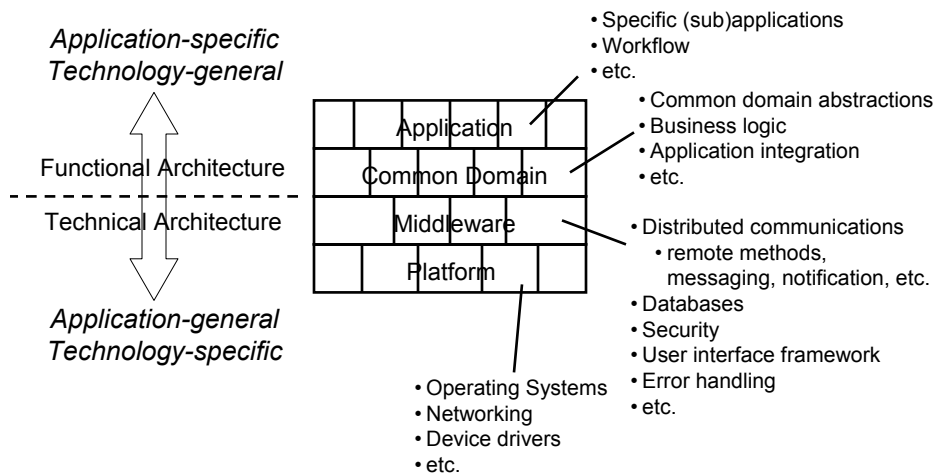
24

Architectural Patterns

- An architectural pattern expresses a fundamental structural organization schema for software systems
 - Predefined subsystems and their responsibilities
 - Rules and guidelines for organizing the subsystems
 - Relationships between subsystems
- Examples of architecture patterns (sometimes called “styles”)
 - Layers
 - Model-View-Controller (separate the user interface from underlying application model, integrated by a controller)
 - Pipes and filters (Unix, signal processing systems, etc.)
 - Shared data (web-based and corporate information systems with user views of a shared RDBMS)

25

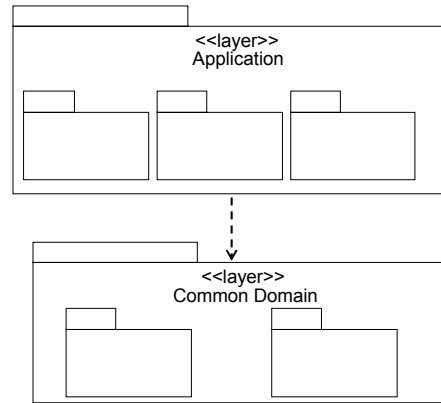
A Layered Architecture



26

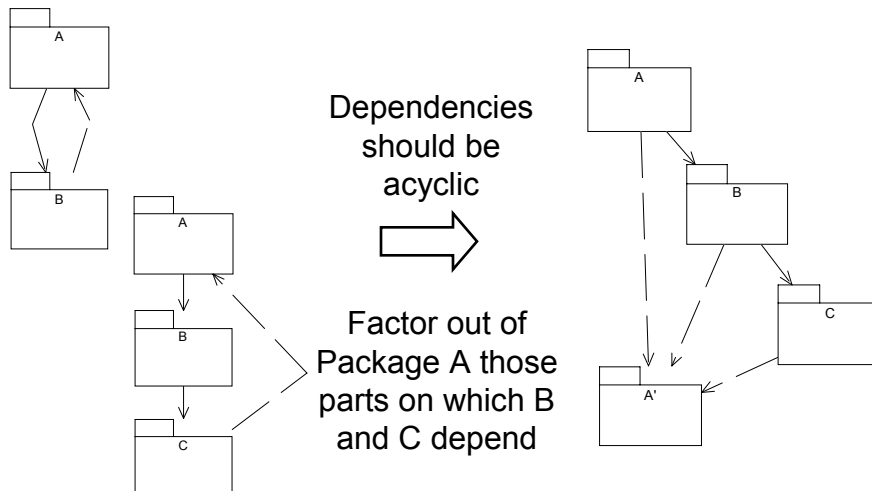
Modeling Architectural Layers

- Think of each layer as a collection of packages dependent only on packages in the layer below
- Use <<layer>> stereotype of a UML package
- Note: frameworks often span layers



27

Layers: Avoid Circular Dependencies



28

Layering Considerations

- Level of abstraction
 - Group elements at the same level of abstraction
- Separation of concerns (cohesion and coupling)
 - Group (“couple”) like (“cohesive”) things together
 - Separate (“decouple”) disparate things
 - Application-specific vs. Domain-common model elements
- Resilience to change
 - Loose coupling (minimal dependencies)
 - Encapsulate things likely to change
 - User interface, business rules, retained data, etc.

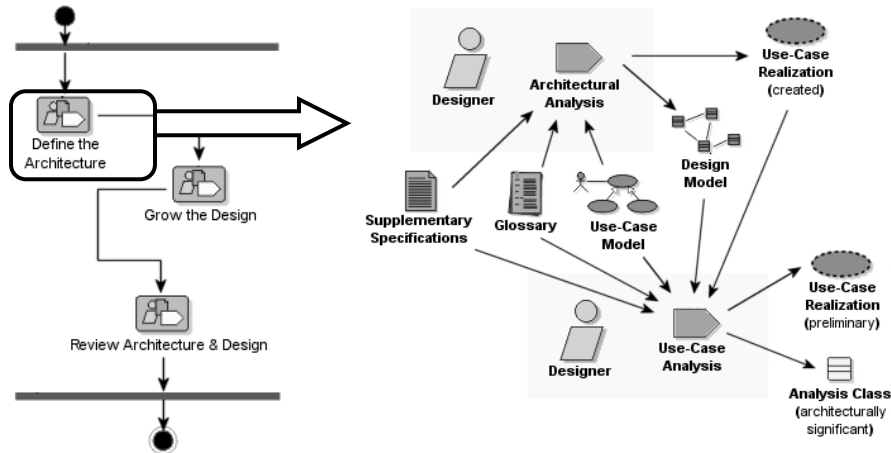
29

Architectural Mechanisms

- An architectural mechanism represents a common solution to a problem occurring a number of places in the system
 - Often a frequently-encountered problem → a pattern
- Provide a clean way to “plug” technology-based solutions into technology-independent application models
- Examples
 - Commercial off-the-shelf products
 - Database management systems
 - Distributed access (networking, remote methods, etc.)
 - Enterprise platforms (Microsoft .NET, Java 2 Enterprise Edition, CORBA Services, etc.)
- Three categories
 - Analysis mechanisms (conceptual: persistence, remote access, etc.)
 - Design mechanisms (concrete: e.g. RDBMS, J2EE)
 - Implementation mechanisms (actual: Oracle, BEA WebLogic)

30

Define the Architecture



31

Define the Architecture

- Purpose
 - Create an initial sketch of the architecture of the system
 - Define an initial set of architecturally significant elements to be used as the basis for analysis
 - Issues that are typically architecturally significant include performance, scaling, process and thread synchronization, and distribution
 - Define an initial set of analysis mechanisms
 - Define the initial layering and organization of the system
 - Define the use-case realizations to be addressed in the current iteration
 - Identify analysis classes from the architecturally significant use cases
 - Update the use-case realizations with analysis class interactions

32

Define the Architecture

(continued)

- How to Staff
 - Small team with cross-functional team members
 - The team should include members with domain experience who can identify key abstractions
 - The team should also have experience with model organization and layering
- Work Guidelines
 - The work is best done in several sessions, perhaps performed over a few days (or weeks and months for very large systems)
 - Iterate between Architectural Analysis and Use-Case Analysis

33

Architectural Analysis

34

Architectural Analysis

- Purpose
 - To define a candidate architecture for the system, based on experience gained from similar systems or in similar problem domains.
 - To define the architectural patterns, key mechanisms and modeling conventions for the system.
 - To define the reuse strategy
 - To provide input to the planning process

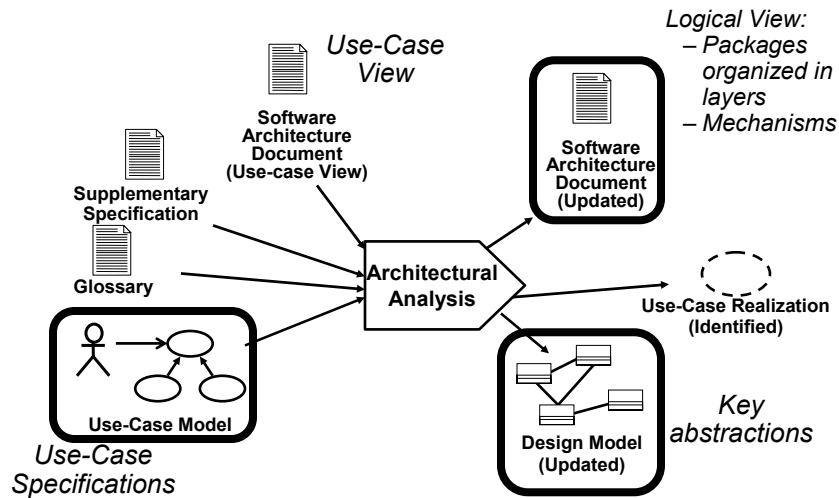
35

Architectural Analysis

- Architectural analysis is an initial attempt to
 - Define the pieces/parts of the system
 - Define the relationships between the pieces/parts
 - Organize the pieces/parts into well-defined layers with explicit dependencies
- A focus on software architecture allows articulation of
 - The structure of the software system
 - Packages, components
 - The ways in which the elements in the structure integrate
 - Mechanisms and patterns of interaction

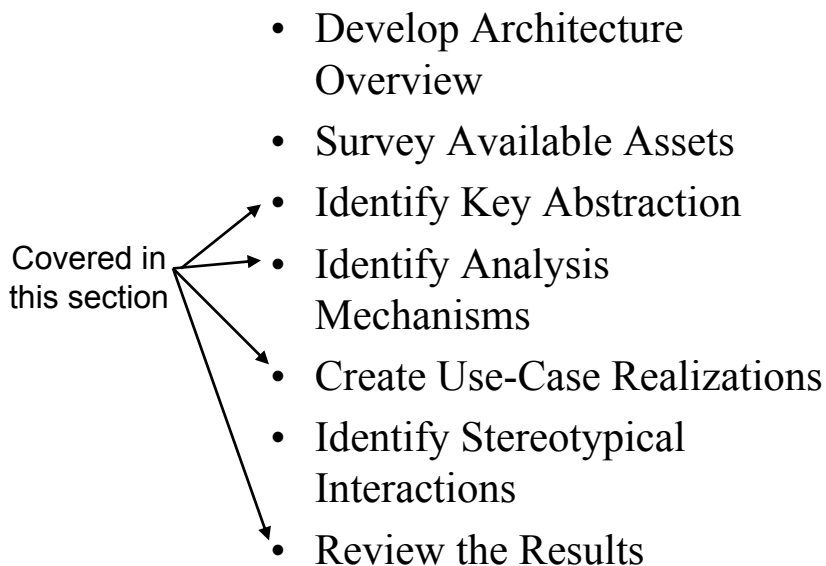
36

Architectural Analysis Activity




37

Architectural Analysis Steps



38

Architectural Analysis Steps

- Develop Architecture Overview
- Survey Available Assets
-  • Identify Key Abstraction
- Identify Analysis Mechanisms
- Create Use-Case Realizations
- Identify Stereotypical Interactions
- Review the Results

39

Key Abstractions

“Prime the Pump” for Analysis

- A key abstraction is a concept, normally uncovered in Requirements, that the system must be able to handle
- Sources for key abstractions
 - Domain knowledge
 - Requirements
 - Glossary
 - Domain model or business model
- Often a concept used in a number of use cases
 - Provides consistency across use-case analysis activities

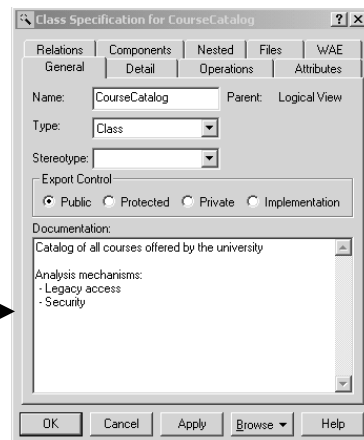
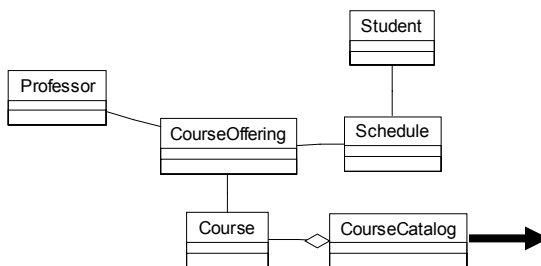
40

Identifying Key Abstractions

- Identify preliminary entity analysis classes to represent these key abstractions on the basis of general knowledge of the system
 - Create a short description for each
- Define any relationships that exist between entity classes
- Present the key abstractions in one or several class diagrams
- Note: The purpose of this step is not to identify a set of classes that will survive throughout design, but to identify the key concepts the system must handle
 - You will find more entity classes and relationships when looking at the use cases
 - Analysis classes will be re-factored and augmented with helper classes in design


41

Key Abstractions: Example



42

Architectural Analysis Steps

- Develop Architecture Overview
- Survey Available Assets
- Identify Key Abstraction
-  • Identify Analysis Mechanisms
- Create Use-Case Realizations
- Identify Stereotypical Interactions
- Review the Results

43

Analysis Mechanisms

- Analysis mechanisms focus on and address the non-functional requirements of the system
 - For example, the need for persistence, reliability, performance, etc.
 - Build support for these requirements directly into the architecture
- In analysis, identify the need for the mechanisms
 - In design, provide the mechanisms
 - Short-hand representation of complex behavior
 - Avoids becoming bogged-down in the specification of relatively complex behavior that is needed to support the functionality but which is not central to it
 - Assures the needs are addressed in a consistent way

44

Sample Analysis Mechanisms

- Persistence
- Communication (between processes and between nodes)
- Message routing
- Distribution (directory services, naming and trader services)
- Common facilities: email, printing, backups, etc.
- Transaction management
- Process control and synchronization (resource contention)
- Information exchange, format conversion
- Security
- Error detection, handling, and reporting
- Redundancy
- Legacy interface

45

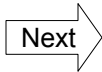
Example Analysis Mechanism Characteristics

- Characterize the need – provide guidance for design choices
 - Persistence
 - Granularity
 - Volume
 - Duration
 - Identification and access mechanism
 - Access frequency
 - Reliability
 - etc.
 - Remote communication
 - Latency
 - Synchronization
 - Message sizes
 - Protocol
 - etc.
 - Security
 - Data granularity (package, class, attribute, etc.)
 - User granularity (Single user, Roles, Groups; Per session or per access; etc.)
 - Security rules
 - Privilege types: read, write, edit, etc.

46

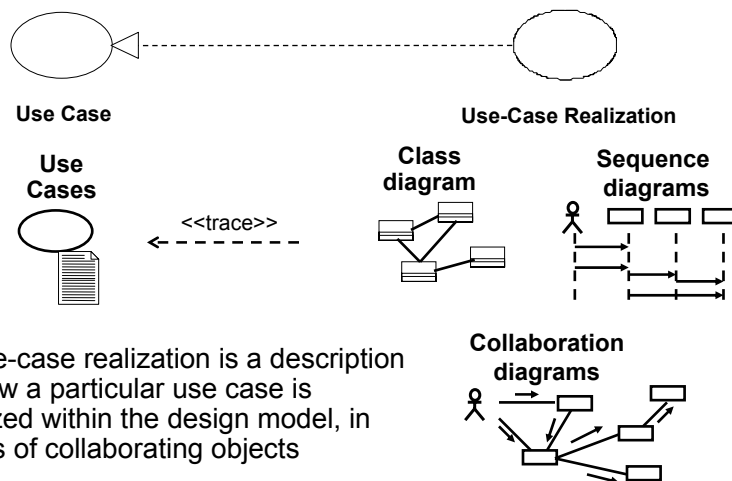
Architectural Analysis Steps

- Develop Architecture Overview
- Survey Available Assets
- Identify Key Abstractions
- Identify Analysis Mechanisms
- Create Use-Case Realizations
- Identify Stereotypical Interactions
- Review the Results



47

Use-Case Realization



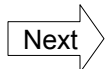
48

Create Use-Case Realizations

- In Architecture Analysis
 - Identify a realization for each use case
 - Create a (blank) class diagram
 - Will contain the classes participating in the use case
 - Create a (blank) collaboration or sequence diagram
 - Will show the classes collaborating to realize the flow of events in the use case

49

Architectural Analysis Steps

- Develop Architecture Overview
- Survey Available Assets
- Identify Key Abstraction
- Identify Analysis Mechanisms
- Create Use-Case Realizations
-  • Identify Stereotypical Interactions
- Review the Results

50

Checkpoints

- General
 - Is the package partitioning and layering done in a logically consistent way?
 - Have the necessary analysis mechanisms been identified?
- Packages
 - Have we provided a comprehensive picture of the services of the upper-level layers?
- Classes
 - Have the key entity classes and their relationships been identified and accurately modeled?
 - Does the name of each class clearly reflect the role it plays?
 - Are the key abstractions/classes and their relationships consistent with the business model, domain model, requirements, glossary, etc.?

51

Part IV

Design Patterns

52

Design Patterns

From

Design Patterns: Elements of Reusable Object-Oriented
Software

By

Erich Gamma, Richard Helm, Ralph Johnson and John
Vlissides (*Gang of Four—GoF*)

53

An Introduction to Patterns

- Here are some quotes Linda Rising uses to describe design patterns
 - Each pattern describes a problem that occurs over and over again in our environment and then describes the core of the solution to that problem in such a way that you can use this solution a million times over without ever doing it the same way twice. Christopher Alexander describing building architecture patterns
 - Patterns provide an incredibly dense means of efficient and effective communication between those who know the language. Nate Kirby
 - Human communication is the bottleneck in software development. If [patterns] can help [developers] communicate with their clients, their customers, and each other, then [patterns] help fill a crucial need in [our industry]. Jim Coplien
 - Patterns don't give you code you can drop into your application, they give you experience you can drop into your head. Patrick Logan
 - Giving someone a piece of code is like giving him a fish; giving him a pattern is like teaching him to fish. Don Dwiggins

Linda Rising, *The Patterns Handbook*, Cambridge Univ Press, 1998

54

Motivation

- Become better designers
 - Design is hard!
- Get exposed to design issues and approaches
- Acquire important design patterns to add to your toolbox

55

Design Patterns

- Simple and elegant solutions to specific problems in object-oriented software design
 - Developed and evolved over time – not your “first” approach
- Designing object-oriented software is hard
- Designing *reusable* object-oriented software is even harder
- Design experts reuse solutions that have worked for them in the past
 - Use good solutions again and again
 - Base new designs on prior experience
- Notes
 - Patterns are a starting point, not a destination
 - Models are not right or wrong, they are more or less useful

56

Designing *Reusable* Object-Oriented Software

- Find objects
- Factor into classes at right granularity
- Define class interfaces
- Define class relationships
- Define inheritance hierarchies
- Make the design specific *and* general
- Re-use it and modify it based on experience
- Notes
 - Build it for the specific use, first; make it reusable through 3+ iterations
 - A common failure is designing for reuse too early
 - Patterns are discovered, not invented
 - See John Vlissides: *Pattern Hatching*

57

Using Design Patterns

- Makes it easier to reuse successful designs and architectures
- Makes proven techniques more accessible to developers
- Helps developer
 - Choose among alternatives
 - Make a system more reusable
 - Avoid alternatives that compromise reusability
 - Improve documentation
 - Get a design “right” faster

58

Design Pattern: Elements

- Pattern name
 - A “handle” to describe a problem, solutions and consequences
 - Vocabulary is a powerful enabler to thinking and communicating
- The problem
 - When to apply the pattern
- The solution
 - Objects and classes; their responsibilities, relationships, and collaborations
 - A template for many situations and implementations
- Consequences
 - Design alternatives -- costs and benefits
 - Language and implementation issues

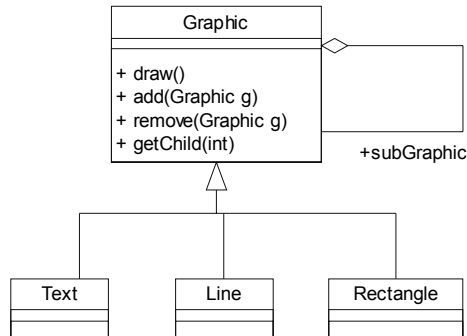
59

Pattern Documentation

- | | |
|---------------------------|-------------------------|
| • Name | • Collaborations |
| • Intent | • Consequences |
| • Motivation and examples | • Implementation Issues |
| • Applicability | • Sample Code |
| • Structure | • Known Uses |
| • Participants | • Related patterns |

60

Example



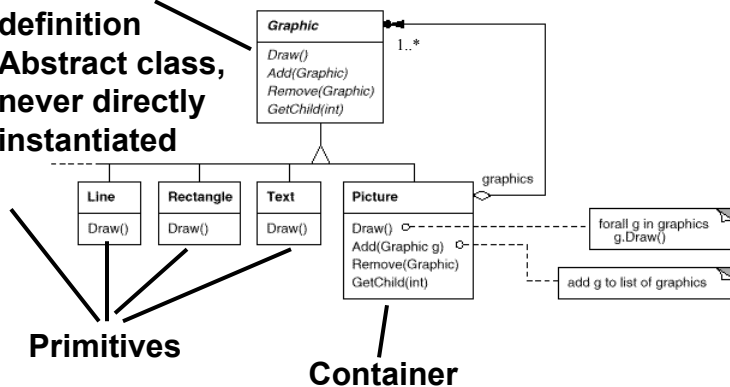
- A client has to treat primitive and container objects differently
- How do add, remove, and getChild behave on primitive graphics (line, text, etc.)?

61

Example Composite: Class Diagram

Key: an abstract class that represents *both* primitives and their containers

- Common definition
- Abstract class, never directly instantiated



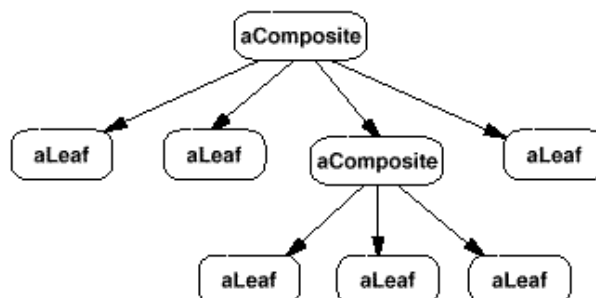
62

Example: Composite Pattern

- Object Structural pattern
- Intent
 - Compose objects into tree structures to represent part-whole hierarchies.
 - Composite lets clients treat individual objects and compositions of objects uniformly.

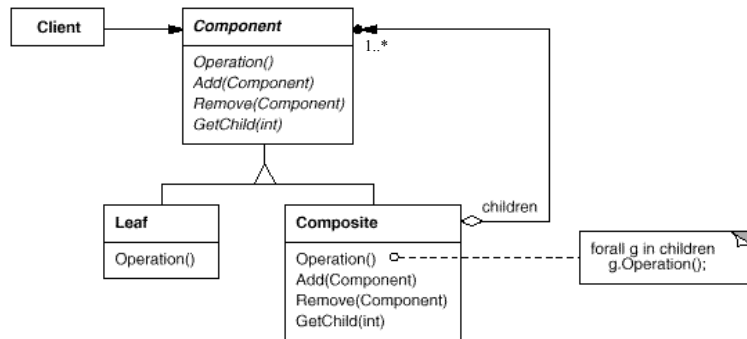
63

Structure: Typical Object Structure



64

The Generic Structure: Class Diagram



Note: OMT notation, a predecessor to UML
(from Rumbaugh, *et al.*; Rumbaugh is one of the UML Three Amigos)

65

Participants

- **Component** (Graphic)
 - declares the interface for objects in the composition.
 - implements default behavior for the interface common to all classes, as appropriate.
 - declares an interface for accessing and managing its child components.
 - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- **Leaf** (Rectangle, Line, Text, etc.)
 - represents leaf objects in the composition. A leaf has no children.
 - defines behavior for primitive objects in the composition.
- **Composite** (Picture)
 - defines behavior for components having children.
 - stores child components.
 - implements child-related operations in the Component interface.
- **Client**
 - manipulates objects in the composition through the Component interface

66

Collaborations

- Clients use the Component class interface to interact with objects in the composite structure
- If the recipient is a Leaf, then the request is handled directly
- If the recipient is a Composite, then it usually forwards requests to its child components
 - Possibly performs additional operations before and/or after forwarding.

67

Consequences

The Composite pattern

- Defines class hierarchies consisting of primitive objects and composite objects.
 - Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively.
 - Wherever client code expects a primitive object, it can also take a composite object.
- Makes the client simple.
 - Clients can treat composite structures and individual objects uniformly.
 - Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component.
 - This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.

68

Consequences

(continued)

The Composite pattern

- Makes it easier to add new kinds of components.
 - Newly defined Composite or Leaf subclasses work automatically with existing structures and client code.
 - Clients don't have to be changed for new Component classes.
- Can make your design overly general.
 - The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite.
 - Sometimes you want a composite to have only certain components.
 - With Composite, you can't rely on the type system to enforce those constraints for you.
 - You'll have to use run-time checks instead.

69

Implementation Issues

1. Explicit parent references.

- Maintaining references from child components to their parent
 - Simplify the traversal and management of a composite structure.
- Parent reference simplifies moving up the structure and deleting a component.
- Parent references help support the Chain of Responsibility pattern.
- The usual place to define the parent reference is in the Component class.
 - Leaf and Composite classes can inherit the reference and the operations that manage it.
- Essential to assure that all children of a composite have as their parent the composite that in turn has them as children.
 - Easiest way: change a component's parent only when it's being added or removed from a composite
 - It can be inherited by all the subclasses, assuring the need automatically.

70

Implementation Issues

(continued)

2. Sharing components.

- Useful to share components, for example, to reduce storage requirements
 - But when a component can have no more than one parent, sharing components becomes difficult.
- A possible solution is for children to store multiple parents.
 - But that can lead to ambiguities as a request propagates up the structure.
- The Flyweight pattern shows how to rework a design to avoid storing parents altogether.
 - It works in cases where children can avoid sending parent requests by externalizing some or all of their state.

71

Implementation Issues

(continued)

3. Maximizing the Component interface.

- Goal of the Composite pattern: make clients unaware of the specific Leaf or Composite classes they're using.
 - Define as many common operations for Composite and Leaf classes as possible.
- Operations that Component supports that don't seem to make sense for Leaf classes. How can Component provide a default implementation for them?
 - For example, the interface for accessing children is a fundamental part of a Composite class but not necessarily Leaf classes.
 - Define a default operation for child access in the Component class that never returns any children.
 - Leaf classes can use the default implementation, but Composite classes will reimplement it to return their children.⁷²

Implementation Issues

(continued)

4. Declaring the child management operations

- Issues
 - Defining child management at Component class provides transparency (treat composite and leaf the same) but costs safety (clients may try to add/remove objects from leaves)
 - Defining child management in Composite class gives safety (add/remove from leaf caught at compile time for statically-typed language like C++) but costs transparency (leaves and composites have different interfaces)
- Pattern, as defined, emphasizes transparency over safety.
 - Default Add and Remove in Component class
 - If maintain parent reference, Remove can remove itself from parent
 - Inherited Leaf::Add probably indicates client bug
 - If opt for safety, may sometimes lose type information and have to convert a component into a composite.

73

Implementation Issues

(continued)

5. Should Component implement a list of Components?

- Define set of children as an instance variable in Component class
- Issue: incurs a space penalty in every leaf, even though a leaf never has children
 - Use only if relatively few children in the structure

6. Child ordering

- For example, front-to-back ordering of graphics; syntactic ordering of parse trees
- Child access and management must carefully manage sequence of children
 - See Iterator pattern

74

Implementation Issues

(continued)

7. Caching to improve performance

- Cache traversal or search information for frequently traversed compositions
 - Cache actual search results or just “short-circuits”
 - For example, Picture class could cache the bounding box of its children to avoid drawing children not in current window
- Component changes require invalidating cache of its parents
 - Components should know their parents
 - Need an interface to tell composites their caches are invalid

8. Who should delete components?

- Make a Composite responsible for deleting its children when the Composite is destroyed
- Exception: immutable, shared Leaf objects

9. Component storage data structure

- Linked list, tree, array, hash table, etc. depending on efficiency
- Composite may have variable for each child. See Interpreter pattern ⁷⁵

Composite pattern

(continued)

- Sample Code
 - Provided in Gamma, et al.
- Known Uses
 - Provided in Gamma, et al.
- Related Patterns
 - Often the component-parent link is used for a Chain of Responsibility.
 - Decorator is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.
 - Flyweight lets you share components, but they can no longer refer to their parents.
 - Iterator can be used to traverse composites.
 - Visitor localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes. ⁷⁶

Pattern Organizations

- Deepens insight into patterns
 - What they do
 - How they compare
 - When to apply them

77

Pattern Classification Categories

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

78

Pattern Classification Categories

- **Purpose**
 - Creational: object creation
 - Structural: composition of classes or objects
 - Behavioral: object interaction; distribution of responsibility
- **Scope**
 - Class patterns
 - Relations between classes and subclasses
 - Static (inheritance) – compile-time
 - Object patterns
 - Object relationships
 - Dynamic – run-time

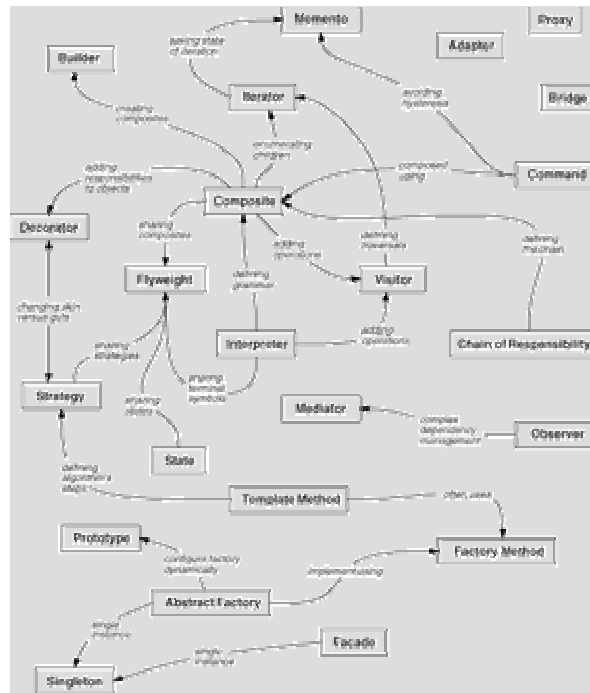
79

Pattern Classification Categories

- **Creational class patterns**
 - Defer some part of object creation to subclasses
- **Creational object patterns**
 - Defer some part of object creation to another object
- **Structural class patterns**
 - Use inheritance to compose classes
- **Structural object patterns**
 - Describe ways to assemble objects
- **Behavioral class patterns**
 - Use inheritance to describe algorithms and flow of control
- **Behavioral object patterns**
 - Describe how a group of objects cooperate to perform a task that no single object can carry out alone

80

Pattern relations



An Essential Strategy: Separation of Concerns

- Separate things that can change separately
- But keep them related so they can cooperate to solve the integrated problem
- Very often the separation results in a level of indirection – an intermediary to coordinate their integrated behavior

A list of Design Patterns (1 of 6)

- **Abstract Factory (object-creational)**
 - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- **Adapter (object-structural)**
 - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Bridge (object-structural)**
 - Decouple an abstraction from its implementation so that the two can vary independently.
- **Builder (object-creational)**
 - Separate the construction of a complex object from its representation so that the same construction process can create different representations.

83

A list of Design Patterns (2 of 6)

- **Chain of Responsibility (object-behavioral)**
 - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- **Command (object-behavioral)**
 - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Composite (object-structural)**
 - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Decorator (object-structural)**
 - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

84

A list of Design Patterns (3 of 6)

- **Facade (object-structural)**
 - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **Factory Method (class-creational)**
 - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Flyweight (object-structural)**
 - Use sharing to support large numbers of fine-grained objects efficiently.
- **Interpreter (class-behavioral)**
 - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

85

A list of Design Patterns (4 of 6)

- **Iterator (object-behavioral)**
 - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Mediator (object-behavioral)**
 - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Memento (object-behavioral)**
 - Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- **Observer (object-behavioral)**
 - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

86

A list of Design Patterns (5 of 6)

- **Prototype (object-creational)**
 - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Proxy (object-structural)**
 - Provide a surrogate or placeholder for another object to control access to it.
- **Singleton (object-creational)**
 - Ensure a class only has one instance, and provide a global point of access to it.
- **State (object-behavioral)**
 - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

87

A list of Design Patterns (6 of 6)

- **Strategy (object-behavioral)**
 - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Template Method (class-behavioral)**
 - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- **Visitor (object-behavioral)**
 - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

88

How Patterns Solve Problems

- Helping in finding objects
- Determining object granularity
- Specifying object interfaces
- Specifying object implementation
 - Specifying an implementation (a class) vs. specifying an interface (a type)
 - Implementation inheritance vs. interface inheritance
 - Client programming to an interface, not an implementation
- Putting reuse mechanisms to work
 - Inheritance vs. composition
 - Delegation
 - Inheritance vs. parameterized types
- Relating compile-time and run-time structures
 - Class definition vs. collaborating objects
- Designing for change (see next slides)

89

Common Causes of Redesign (1/3)

*Common theme: Separation of concerns
Indirection or intermediary decouples specific dependence*

- Creating an object by specifying a class explicitly
 - Commits to implementation instead of a particular interface
 - Complicates future changes
 - Avoid issue by creating objects indirectly
 - Patterns: Abstract Factory, Factory Method, Prototype
- Dependence on specific operations
 - Specify an operation → commit to one way to satisfy request
 - Avoid hard-coded requests → easier to change at compile- and run-time
 - Patterns: Chain of Responsibility, Command

90

Common Causes of Redesign (2/3)

- **Dependence on hardware and software platform**
 - Using operating system API limits portability and ability to migrate to new versions of same platform
 - Limit platform dependencies
 - Patterns: Abstract Factory, Bridge
- **Dependence on object representations or implementations**
 - Clients that know how an object is represented, stored, located, or implemented might need to be changed when the object changes
 - Hiding this information from clients keeps changes from cascading
 - Patterns: Abstract Factory, Bridge, Memento, Proxy
- **Algorithmic dependencies**
 - Algorithms are extended, optimized and replaced
 - Isolate algorithm to minimize object change
 - Patterns: Builder, Iterator, Strategy, Template Method, Visitor

91

Common Causes of Redesign (3/3)

- **Tight coupling**
 - Use abstract coupling and layering to promote loosely coupled systems
 - Patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Façade, Mediator, Observer
- **Extending functionality by subclassing**
 - Use object composition and delegation to combine behavior
 - Use patterns that define a simple subclass and compose its instances with existing ones
 - Patterns: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy
- **Inability to alter classes conveniently**
 - For example, don't have source code (library) or changes would require modifying existing subclasses
 - Use patterns that modify classes
 - Patterns: Adapter, Decorator, Visitor

92

How to Select a Design Pattern

- Consider how design patterns solve design problems
- Scan Intent sections of pattern documentation
 - See later slides
- Study how patterns interrelate
 - See pattern relations diagram
- Study patterns of like purpose
 - Understand similarities and differences of patterns grouped together in pattern categories table
- Examine a cause of redesign
 - See discussion of causes of redesign
- Consider what should be variable in your design
 - Encapsulate the concept that varies instead of redesign
 - Understand what design aspects that patterns let you vary

93

How to Use a Design Pattern

- Read the pattern once through for an overview
- Go back and study the structure, participants and collaborations sections
- Look at the sample code to see a concrete example of the pattern in code
 - Learn how to implement the pattern
- Choose names for pattern participants that are meaningful in the application context
- Define the classes
 - Including identification of existing classes in application that the pattern will affect
- Define application-specific names for operations
- Implement the operations to carry out responsibilities and collaborations in the pattern

94

Some Other Pattern Stuff

- Analysis Patterns, by Martin Fowler
 - Groups of concepts that represent common constructs in business modeling
 - Examples
 - Accountability and organizational structure
 - Observations and measurements
 - Observations for corporate finance
 - Inventory and accounting
 - Planning and monitoring actions
 - Trading
 - Contracts
- Buschmann Architecture Patterns ---
http://www.rationalrose.com/models/architectural_patterns.htm
- Anti-Patterns ---
<http://www.serve.com/mowbray/ICSE.zip>
- Refactoring patterns ---
<http://www.refactoring.com/>

95

Other Design Patterns Stuff

- <http://www.hillside.net/patterns> --- Hillside Group
- <http://c2.com/ppr> --- Portland Pattern Repository
- Pattern Languages of Programming conferences (PLoP)
- <http://www.agcs.com/supportv2/techpapers/patterns/papers/tutnotes/index.htm> --- AGCS examples
 - Some non-software examples of the GoF patterns to help build intuition about the patterns
- Patterns support built into IBM/Rational Rose, Borland/Together Control Center, and other modeling/IDEs

96

Part V

Architectural Patterns

97

Suggested Readings

- Read the following sections of the GoF book (Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995)
 - Section 1.2: Design Patterns in Smalltalk MVC
 - Section 1.3: Describing Design Patterns
 - Section 1.4: The Catalog of Design Patterns
 - Section 1.5: Organizing the Catalog
 - Chapter 2: A Case Study: Designing a Document Editor
- An interactive, web-based version of this classic book is available on the Web ([file:///Hephaestus/DFS-Software/SW/Engineering Tools/Design Patterns/index.htm](file:///Hephaestus/DFS-Software/SW/Engineering%20Tools/Design%20Patterns/index.htm))

98

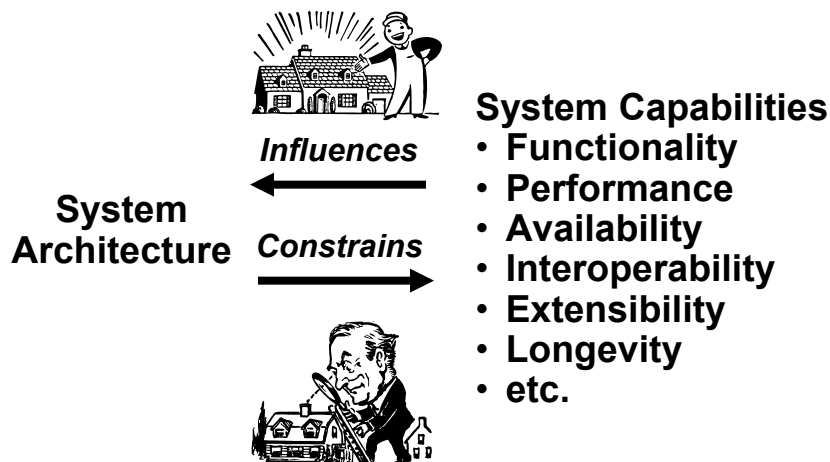
Software Architecture

Sources

- Len Bass, Paul Clements, Rick Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.
- Ian Sommerville, *Software Engineering* 6th Ed., Addison-Wesley, 2001.
- Frank Buschmann *et al.* *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.
- Jan Bosch, *Design and Use of Software Architectures: Adopting and evolving a product-line approach*, ACM Press-Addison Wesley, 2000.
- The Rational Unified Process

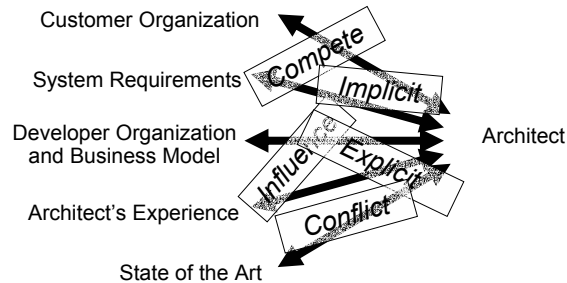
99

Architecture is the Foundation for a Successful System



100

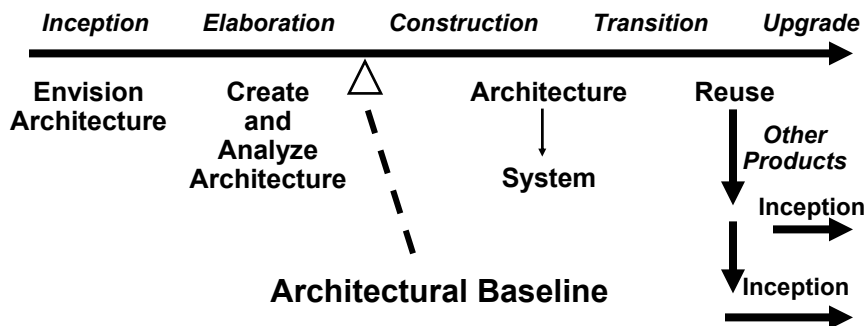
Intricate Waltz of Influence and Counterinfluence



- Architects must identify and actively engage the stakeholders to solicit their needs and expectations
 - Technically Correct AND Politically Correct
- A software architect must have considerable organizational talents and negotiating skills
 - In addition to comprehensive technical and domain knowledge

101

Architecture in the Product Life-Cycle



102

Architecture: Pieces, Parts, Relations, Context

- The kinds of components that are used in developing a system according to the pattern
- The connectors, or kinds of interactions among the components
- The control structure or execution discipline
- The underlying intuition behind the pattern, or the system model

From Mary Shaw, “Some Patterns for Software Architectures,” *Pattern Languages of Program Design, Vol 2*, John Vlissides, James Coplien, Norman Kerth (eds), Addison-Wesley 1996, pp. 255-269.

103

Implications

- Architecture is an abstraction of systems
 - Focus on components and interaction
 - Suppresses local information: component details are not architectural
- Systems have many structures (views)
 - No single view can be the architecture
 - More than one kind of component (e.g., module or process), interaction (e.g., subdivision or synchronization) and/or context (e.g., build-time or run-time)
 - Use whatever views are useful for analysis or communication

104

Implications

(continued)

- Every system has an architecture
 - Is it known?
 - Is it engineered?

Software architecture is a focus on reasoning about the structural issues of a system

105

Big Ball of Mud Architecture Style – Abstract

<http://www.laputan.org/mud/mud.html>

While much attention has been focused on high-level software architectural patterns, what is, in effect, the de-facto standard software architecture is seldom discussed. This paper examines this most frequently deployed of software architectures: the BIG BALL OF MUD. A BIG BALL OF MUD is a casually, even haphazardly, structured system. Its organization, if one can call it that, is dictated more by expediency than design. Yet, its enduring popularity cannot merely be indicative of a general disregard for architecture.

These patterns explore the forces that encourage the emergence of a BIG BALL OF MUD, and the undeniable effectiveness of this approach to software architecture. What are the people who build them doing right? If more high-minded architectural approaches are to compete, we must understand what the forces that lead to a BIG BALL OF MUD are, and examine alternative ways to resolve them.

106

Big Ball of Mud – Introduction

A BIG BALL OF MUD is *haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle*. We've all seen them. These systems show unmistakable signs of *unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated*. The overall *structure of the system may never have been well defined*. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.

Still, *this approach endures and thrives*. Why is this architecture so popular? Is it as bad as it seems, or might it serve as a way-station on the road to more enduring, elegant artifacts? What forces drive good programmers to build ugly systems? Can we avoid this? Should we? How can we make such systems better?

107

Some Patterns that Result in Big Balls of Mud

- Throw-away prototype code that is not thrown away
- Piecemeal growth resulting in uncontrolled sprawl
- Keeping it working: no time to go back and fix the architecture
- Sweeping under the rug (put a façade around messy parts)

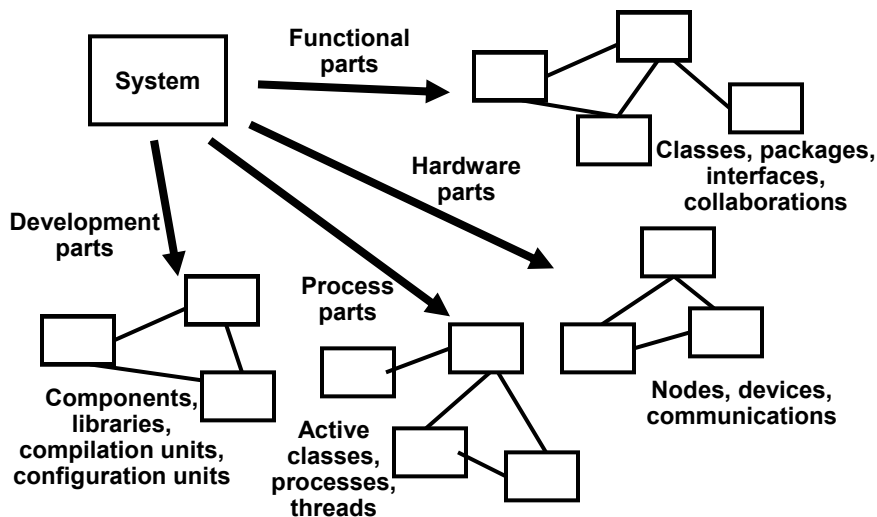
108

Why Architecture?

- Communication and consensus among stakeholders
 - Common language to address different concerns
- Early design decisions and earliest point of analysis
 - Constrains implementation
 - Dictates structure of development organization
 - Inhibits/enables system quality attributes
 - Enables analysis and reasoning about quality and change
 - Skeleton for evolutionary development
- Transferable abstraction of a system
 - More value if reuse requirements, then architecture, then design, then code
 - Product lines, components, designs, interaction templates
 - Basis for training

109

Decomposition into Parts



110

Architectural Structures

- **Module/sub-module structure**
 - Units: work assignments
 - Relationships: sub-module of; shares secret with
- **Conceptual (logical, functional) structure**
 - Units: functions
 - Relationships: shares data with
- **Process and coordination structure**
 - Units: programs
 - Relationships: runs concurrently with; may run concurrently with; excludes; precedes, etc. (synchronization and run-time dependence)
- **Physical structure**
 - Units: hardware
 - Relationships: communicates with

111

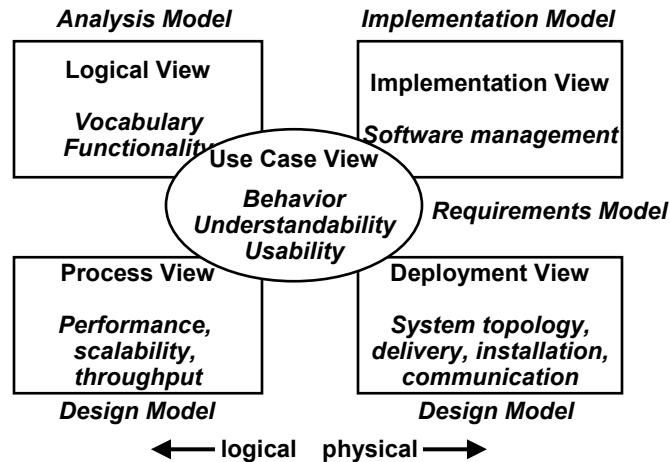
Architectural Structures

(continued)

- **Uses structure**
 - Units: programs
 - Relations: functional dependency; requires the presence of
- **Calls structure**
 - Units: programs
 - Relationships: invokes with parameters (execution flow)
- **Data flow**
 - Units: functional tasks
 - Relationships: may send data to
- **Control flow**
 - Units: system states or modes
 - Relationships: transitions to, subject to the events and conditions labeling the link
- **Class structure**
 - Units: objects
 - Relationships: is an instance of; shares access methods

112

4+1 Views of Architecture *and the models that define the views*



113

Buschmann et al.

A Pattern Classification Schema

		Scope →		
Purpose ↓		Architectural Patterns	Design Patterns	Idioms
	From Mud to Structure	Layers Pipes and Filters Blackboard		
	Distributed Systems	Broker Pipes and Filters Microkernel		
	Interactive Systems	MVC PAC		
	Adaptable Systems	Microkernel Reflection		
	Structural Decomposition		Whole-Part	
	Organization of Work		Master-Slave	
	Access Control		Proxy	
	Management		Command Processor View Handler	
	Communication		Publisher-Subscriber Forwarder-Receiver Client-Dispatcher-Server	
	Resource Handling			Counted Pointer

114

Claims

- Architecture is the development product that gives the highest return on investment (ROI) with respect to quality, schedule and cost
 - Relatively inexpensive to check and fix, upstream
 - Substantial downstream consequences
- Reusable components need architectural context
- Architectures are also reusable
- Conceptual integrity is the key to sound system design
 - conceptual integrity can only be had by a small number of minds coming together to design the system's architecture [Brooks]

115

What Makes a “Good” Architecture

Some Process Guidelines

- Single architect or small group with identified leader
- Must have system technical requirements and articulated, prioritized qualitative properties
- Architecture must be well-documented using an agreed-on notation that all can understand
- Architecture should be actively reviewed by all stakeholders
- Analyze architecture for applicable quality measures and formally review early in process
- Architecture should allow creation of a skeletal system on which functionality can incrementally grow
- Architecture should result in specific resource restriction/allocation models (memory, bandwidth, time)

116

What Makes a “Good” Architecture

Some Product (Structural) Guidelines

- Well-defined functional modules using information hiding, separation of concerns, and well-defined interfaces
- Module separation of concerns should allow concurrent, relatively independent development by separate teams
- Modules should hide idiosyncrasies of the computing infrastructure and platform
- Never depend on a particular version of a commercial product or tool; make change straightforward and inexpensive
- Separate data producers from data consumers
- Parallel processing: well-defined processes and tasks that may not mirror module structure
- Process or task assignment to processors must be easily changed, even at run-time
- Consistently use a small number of simple interaction patterns

117

Common Misconceptions About Software Architecture

- | | |
|---|--|
| <ul style="list-style-type: none">• “Architecture is design”<ul style="list-style-type: none">– Yes, but not all design is architecture• “Architecture is infrastructure”<ul style="list-style-type: none">– Yes, infrastructure is an important part of architecture, but architecture is whole system, not just foundation• “Architecture is [insert favorite technology here]”<ul style="list-style-type: none">– [network, database, GUI, CORBA, .NET, J2EE, ...]– There is more to architecture than technology | <ul style="list-style-type: none">• “Architecture is the work of a single architect”<ul style="list-style-type: none">– Yes, if the architect is a genius, but in practice architecture is the work of a small team• “Architecture is flat (shown using one kind of concept and one family of components and connectors)”<ul style="list-style-type: none">– Complex architecture has multiple views—separate but interdependent structures |
|---|--|

(continued)

118

Common Misconceptions About Software Architecture

(continued)

- “Architecture is structure”
 - Yes, but architecture must also deal with dynamic issues that cross structure—message/event flow, protocols, machine states, thread creation, etc.
 - Architecture must also address “fit” into context of user needs and developer abilities
- “System architecture precedes software architecture”
 - Yes, you need some system context, but system, hardware, and software architecture are developed concurrently and iteratively
- “Architecture cannot be measured or validated”
 - Yes, to validate that an architecture will work, you have to implement it and try it
 - But there are many aspects you can validate by inspection, modeling, simulation, systematic analysis

(continued)

Philippe Kruchten, *The Rational Edge*, April, 2001

119

Common Misconceptions About Software Architecture

(continued)

- “Architecture is a science”
 - Not yet. Scientific, analytical studies are hard to apply to something that is not ridiculously small
- “Architecture is an art”
 - The artistic, creative part of architecture is typically small
 - Most architectures are combinations of known good solutions

Philippe Kruchten, *The Rational Edge*, April, 2001

120

IEEE Std 1471: Recommended Practice for Architectural Description of Software- Intensive Systems

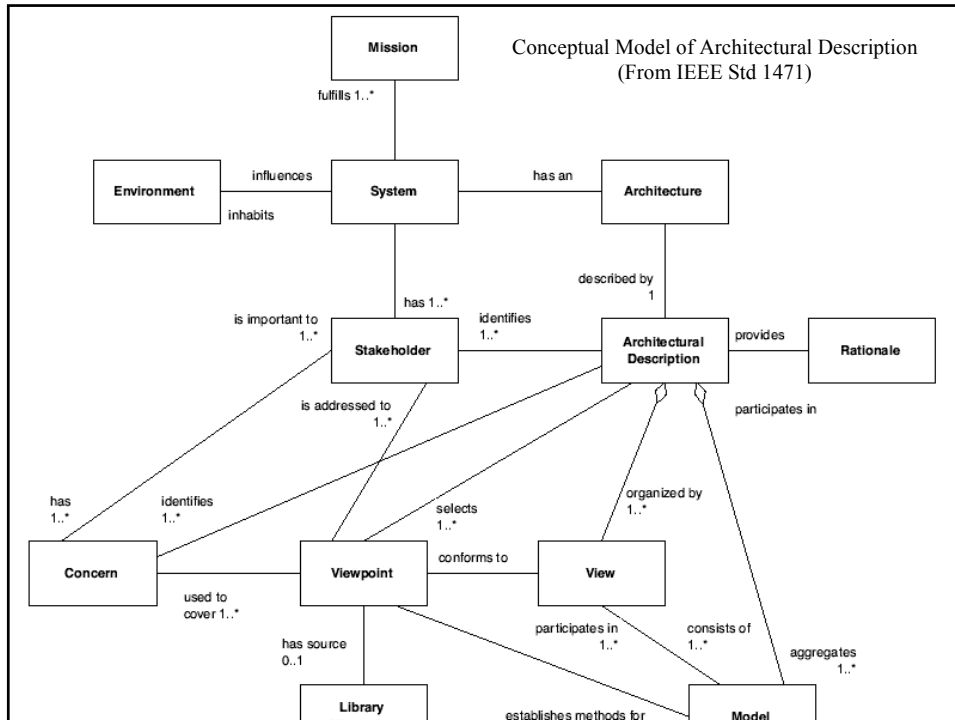
121

Terminology

From IEEE Std 1471: Recommended Practice for Architectural Description of Software-Intensive Systems

- Architecture: The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. (Product)
- Architecting: The activities of defining, documenting, maintaining, improving, and certifying proper implementation of an architecture. (Process)
- Architect: The person, team, or organization responsible for systems architecture. (People)
- View: A representation of a whole system from the perspective of a related set of concerns.
- Viewpoint: A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.

122



Architectural Viewpoints

IEEE STD 1471-2000: "Architectural Descriptions"

- Each viewpoint shall be specified by
 - A viewpoint name
 - The stakeholders to be addressed by the viewpoint
 - The concerns to be addressed by the viewpoint
 - The language, modeling techniques, or analytical methods to be used in constructing a view based on the viewpoint
 - The source, for a library (reused) viewpoint
- A viewpoint may include additional information, such as
 - Consistency and completeness tests
 - Evaluation or analysis techniques
 - Heuristics, patterns, or other guidelines
- Each view in an architectural description shall conform to exactly one viewpoint

Architectural Descriptions

- IEEE STD 1471-2000: “Architectural Descriptions” does not require any specific viewpoints
 - The “correct” viewpoints are driven by the needs of the stakeholders and the system requirements.
- However, there are some viewpoints that are commonly used
 - Three of them follow
 - Structural viewpoint
 - Behavioral viewpoint
 - Physical interconnection viewpoints
- There are many other viewpoint possibilities

125

Structural Viewpoint

- Examples
 - Processing elements, data elements, connectors
 - Components, connectors, structural organization, constraints
- Concerns
 - What are the computational elements of a system and the organization of those elements?
 - What software elements compose the system?
 - What are their interfaces?
 - How do they interconnect?
 - What are the mechanisms for interconnection?

126

(continued)

Structural Viewpoint

(continued)

- Language
 - *Components* (individual units of computation)
 - Components have *ports* (interfaces)
 - *Connectors* (represent interconnections between components for communication and coordination)
 - Connectors have *roles* (where they attach to components)
 - Components and connectors may be typed
 - All of the previously listed entities may have *attributes* of varying kinds

(continued)

127

Structural Viewpoint

(continued)

- Analytic methods
 - The structural viewpoint supports the following kinds of checking:
 - Attachment (are connectors properly connecting components?)
 - Type consistency (are the types of components and connectors used consistent with their attachments and any style or other constraints?)

128

Behavioral Viewpoint

- Concerns
 - What are the dynamic actions of and within a system?
 - What are the kinds of actions the system produces and participates in?
 - How do those actions relate (ordering, synchronization, etc.)?
 - What are the behaviors of system components? How do they interact?
- Modeling methods
 - Events, processes, states, and operations on those entities

129

Behavioral Viewpoint

(continued)

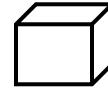
- Analytic methods
 - Some examples are: communicating sequential processes, the pi-calculus, and partially ordered sets of events

130

Physical Interconnect Viewpoint

- Concerns

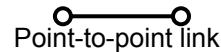
- What are the physical communications interconnects and their layering among system components?
- What is the feasibility of construction, compliance with standards, and evolvability?



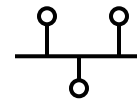
Physically identifiable node

- Viewpoint language

- Define the system with a set of one or more block diagrams using the following visual glossary
- One diagram is provided for each identified communication layer. Each diagram is annotated to show the addressing, link access, and routing/switching strategies at each link.



Point-to-point link



Shared link

131

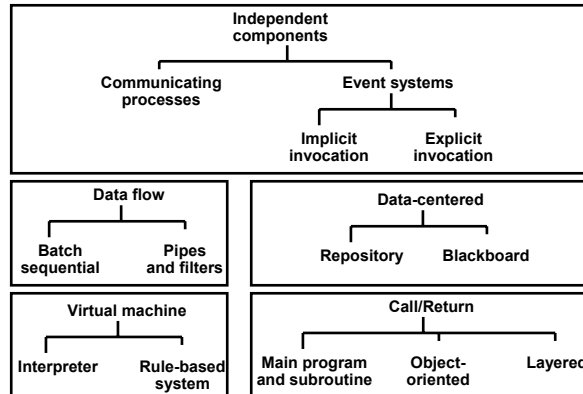
Styles

Moving from Qualities to Architectures

- Architectural styles help software engineers reason about architectural qualities
- A style
 - describes a class of architectures
 - is found repeatedly in practice
 - is a package of design decisions
 - has known properties that permit reuse

132

Architecture Styles



- A growing catalog of styles (this one is from Garlan and Shaw, 1995)
- No complete list
- Styles overlap
- Systems exhibit multiple styles at once

133

Architectural styles

- The architectural model of a system may conform to a generic architectural model or style
- An awareness of these styles can simplify the problem of defining system architectures
- Most large systems are heterogeneous and combine single architectural styles

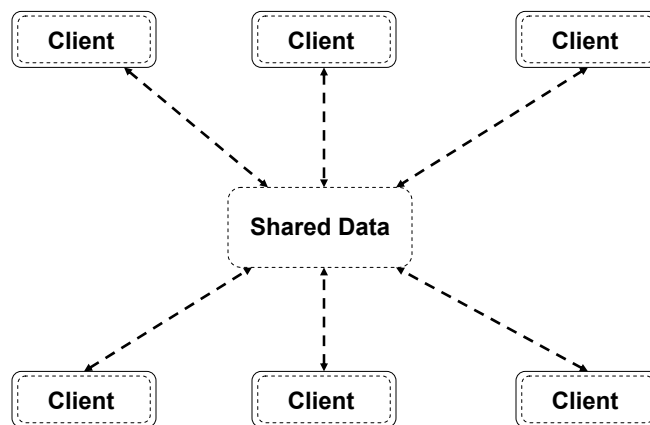
134

Architecture Styles

- A style is described by
 - a set of component types (e.g., data repository, process, object)
 - a set of connector types/interaction mechanisms (e.g., subroutine call, event, pipe)
 - including control structure
 - a topological layout of these components
 - a set of constraints on topology and behavior
 - an *informal* description of the costs and benefits of the style
- A style is named to reflect the underlying intuition behind the pattern or the system model

135

Data-Centered Style



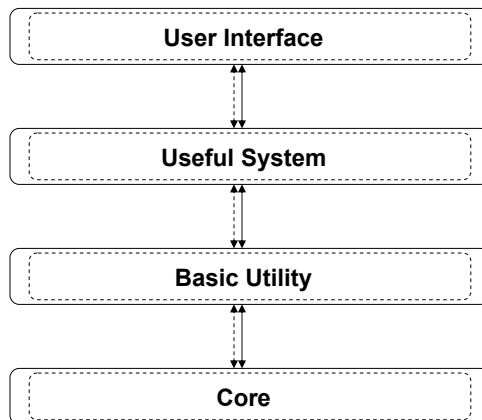
136

Data Centered Style

- Goals
 - integrability
 - modifiability
 - scalability (add new clients and data easily)
- Examples
 - passive data store: repository style
 - active data store: blackboard
- What do we know? What can we ask?
 - How are clients activated?
 - How is the data repository activated?
 - Does the repository know the existence of the clients? Their names? Their locations?
 - Do the repository and client share knowledge of the data representation?

137

Layered Architecture



This is just one example

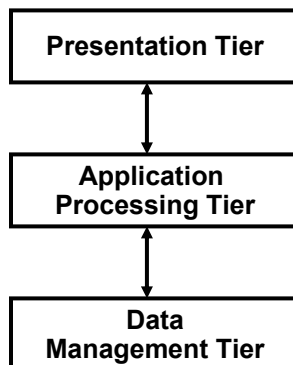
138

Layered Architectures

- Goals
 - portability
 - reuse
 - separation of concerns
- Examples
 - user interfaces
 - telecommunications
 - almost every reasonably complex architecture
- What do we know? What can we ask?
 - How are layers defined and enforced?
 - Can layer elements make calls up, down, sideways?
 - Can layers access non-adjacent layers?
 - Do the elements of a layer share common calling conventions and protocols?

139

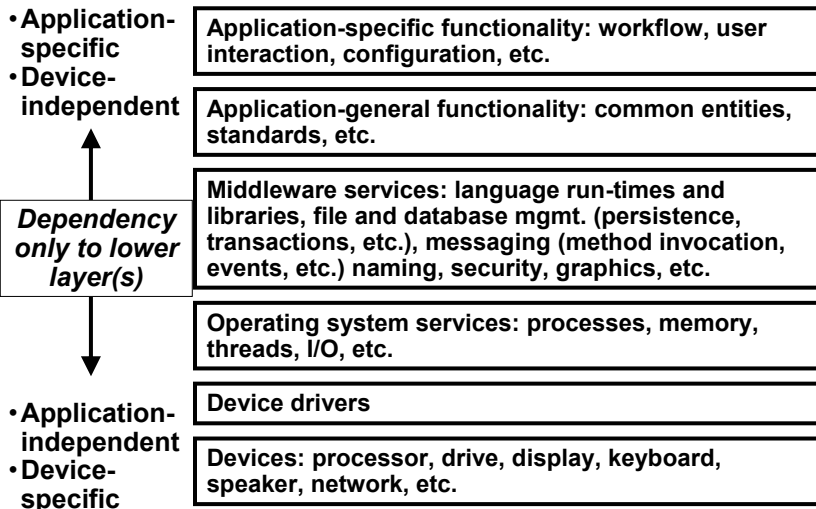
Three-tier Application Architecture



- Presentation tier
 - Concerned with presenting the results of a computation to system users and with collecting user inputs
- Application processing tier
 - Concerned with providing application specific functionality e.g., in a banking system, banking functions such as open account, close account, etc.
- Data management tier
 - Concerned with managing the system databases

140

Layers – an Example

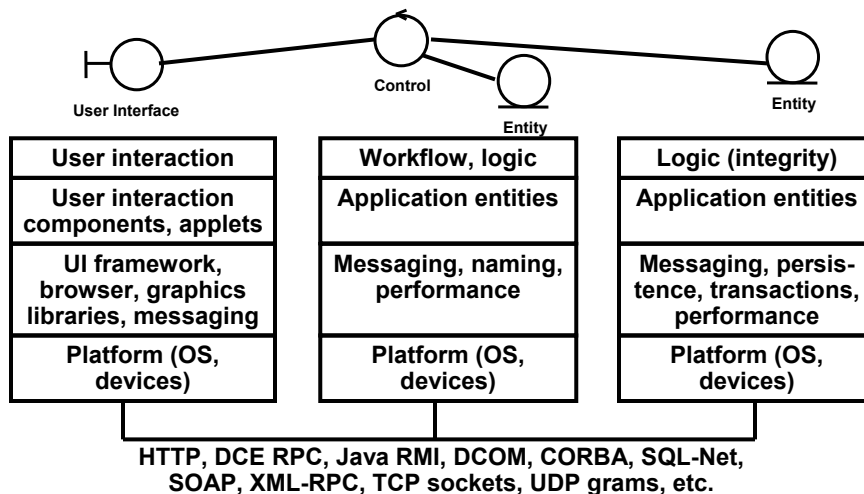


141

Tiers

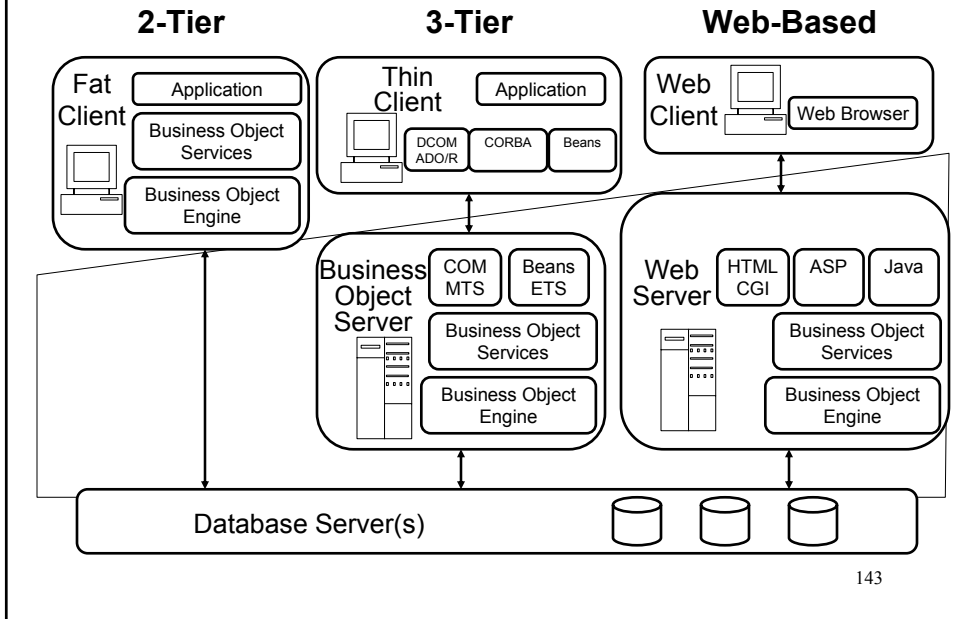
Notice that each tier has layers

- Tiers are layers using one strategy
- Layers of the previous slide use a separate strategy



142

Client/Server Architectures



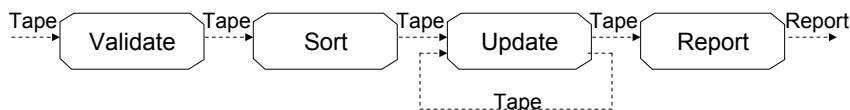
*Next, look at some data
flow architecture styles
in some detail*

Data-Flow Architectures

- Description
 - series of transformations on successive pieces of input data; explicit pattern of data flow
 - availability of data controls the computation
- Goals
 - reuse
 - modifiability
- Examples
 - batch sequential
 - pipe-and-filter
 - process control
- What do we know? What can we ask?

145

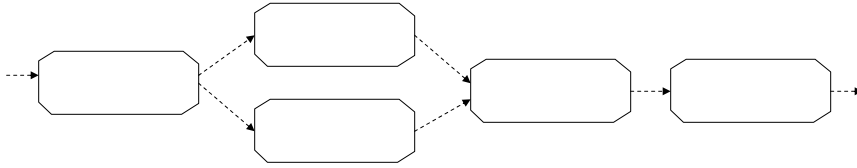
Batch Sequential Style



- Example: Classical data processing
 - typically, nearly linear flow or highly-constrained cyclic
- Each step is an independent program
- Each step runs to completion before the next step starts
- Each batch of data is transmitted as a whole between steps

146

Pipe-and-Filter Style



- Examples: Unix command shell, compilers, signal processing, data-flow, functional programs
- Incremental transformation of data by successive components
 - enrich data, refine data, transform data
- Filter: stream transducer (transformer)
 - context free
 - no state between instantiations
 - no knowledge of upstream or downstream filters
- Pipe: stateless data stream
 - source end feeds filter input, sink receives output

147

Pipe-and-Filter Assessment

- Advantages
 - simplicity: functional composition, no function interaction
 - maintenance and reuse
 - filters are independent black boxes
 - can represent composite of filters and pipes as coarse-grained filter – hierarchy
 - performance: distribute filters to concurrent processors
- Disadvantages
 - interactive applications are difficult (batch mindset)
 - fixed filter ordering; no filter cooperation
 - performance
 - common denominator data representation (e.g., ASCII) or data parse/unparse for transformation
 - input buffers and latency

148

Batch Sequential vs. Pipe and Filter

- Both
 - decompose task into fixed sequence of computations
 - interact only through data passed from one to another

Batch Sequential

- Coarse-grained, total
- High-latency
- Random access to input is OK
- No concurrency
- Non-interactive

Pipe and Filter

- Fine-grained, incremental
- Results start immediately
- Processing localized to input
- Feedback loops possible
- Often interactive
 - but awkward to do so

149

Control Flow vs. Data Flow

- Control Flow (e.g., procedural systems)
 - focus on how center of control moves through system
 - data may accompany the control, but is secondary
 - reason about order of computation
- Data flow
 - focus on how data moves through a collection of computations
 - control is activated by availability of data
 - reason about data availability, transformations, latency, etc.

150

Push vs. Pull

- What “force” makes the data flow?
- Push: source filter pushes data downstream
- Pull: sink filter pulls data from upstream
 - note: control request flows upstream of data
- Push/pull: a filter actively pulls from upstream and pushes downstream
 - combinations may need synchronization if more than one filter is pushing/pulling
 - may have separate control management

151

Enabling Techniques for Software Architecture

From

Frank Buschmann *et al.* *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996, pp.397-404

152

Fundamental Principles of Software Construction: Enabling Techniques

- Abstraction
- Encapsulation
- Information Hiding
- Modularization
- Separation of Concerns
- Coupling and Cohesion
- Sufficiency, Completeness and Primitiveness
- Separation of Policy and Implementation
- Separation of Interface and Implementation
- Single Point of Reference
- Divide-and-Conquer

153

Notes

- Most principles are closely related – complementary
- Some principles contradictory
 - e.g., separate interface and implementation vs. single point of reference

154

Abstraction

- G. Booch: “The essential characteristics of an object [component] that distinguish it from all other kinds of object [components] and thus provide crisply defined conceptual boundaries relative to the perspective of the viewer.”
- Forms (*cf* Seidewitz and Stark, 1986)
 - Entity abstraction
 - Action abstraction
 - Virtual Machine abstraction
 - Co-incidental abstraction
- Patterns
 - Layers
 - Abstract Factory

155

Encapsulation

- Group the elements of an abstraction that constitute its structure and behavior
- Separate different abstractions from each other
- Explicit barriers between abstractions
- Enhances changeability and reusability non-functional requirements

156

Information Hiding

- Conceal the details of a component's implementation from its clients
 - Hides complexity
 - Minimizes coupling
- Mechanisms
 - Encapsulation
 - Separation of interface and implementation
- Reflection relaxes information hiding
 - Opens the implementation in a defined way
 - Provides flexibility for adaptation and change

157

Modularization

- Identify physical packaging of entities that form the logical structure of the system
 - Decompose and group software system into subsystems and components
 - Closely related to encapsulation principle
- Patterns
 - Layers
 - Pipes and Filters
 - Whole-Part

158

Separation of Concerns

- Separate different or unrelated responsibilities
 - e.g., attach them to different components
 - e.g., if a component plays different roles in different contexts, separate those roles in the component (or separate the component)
- A foundational, pervasive principle
- Patterns
 - Almost every patterns uses this principle
 - Model-View-Controller, for example

159

Coupling and Cohesion

- Coupling: inter-module (between)
 - A measure of the “strength” of a connection from one module to another
 - Hard to understand, change, or correct one module without understanding coupled modules
- Cohesion: intra-module (within)
 - A measure of the degree of connectivity between the elements/functions of a single module
 - Functional cohesion – work together to provide some well-bounded behavior
 - Coincidental cohesion – entirely unrelated elements ‘just happen to be’ in same module
 - Others: logical, temporal, procedural, communicational, sequential, informal [H. Balzert 85]
- Patterns: all those that organize communication
 - Client-Dispatcher-Server, Publisher-Subscriber

160

Sufficiency, Completeness and Primitiveness

- Booch: Every component should be sufficient, complete, and primitive
 - Sufficient: captures enough characteristics necessary for a meaningful and efficient interaction with the component
 - Completeness: captures all relevant characteristics
 - Primitive: can easily implement component operations

161

Separation of Policy and Implementation

- A policy component deals with
 - Context-sensitive decisions
 - Knowledge about the semantics and interpretation of information
 - Assembly of disjoint computations into a result
 - Selection of parameter values
- An implementation component deals with
 - A fully-specified algorithm with no context-sensitive decisions
- Implementation: more reusable
- Policy: application-specific, change often

162

Separation of Interface and Implementation

- Two parts of any component
 - Interface: declares the functionality and how to use it
 - Accessible to clients
 - Implementation: code that realizes functionality declared at interface, plus additional functions and data for internal use
- Protects clients from details
- Isolates client from implementation change
- Like encapsulation: ‘a client should only know what it needs to know’

163

Single Point of Reference

- Declare and define an item only once
 - Avoid inconsistency
- Note: C++, etc. defines exactly once, but declares many times
 - Due to limitations of traditional compiler and linker technology
 - Results in increased programmer effort to maintain consistency

164

Divide and Conquer

- Divide a task or component into smaller parts that can be designed independently
- Patterns
 - Whole-Part and other subdivision patterns
 - Microkernel: minimal functional core that is extended and specialized

165

Unit Operations: Atomic Operations on Architectures

- Separation: place a distinct piece of functionality into a distinct component that has a well-defined interface to the rest of the world
 - Decomposition: separate a large component into two or more smaller ones
 - Part-whole
 - Is-a (specialization)
 - Replication: duplicate a component (for fault tolerance and performance)
- Abstraction: create a virtual machine (that hides its underlying implementation)
- Compression: remove layers or interfaces that separate system functions (the opposite of separation)
- Resource sharing: encapsulate data or services and share them among multiple independent consumers

166

From Bass, Clements, Kazman

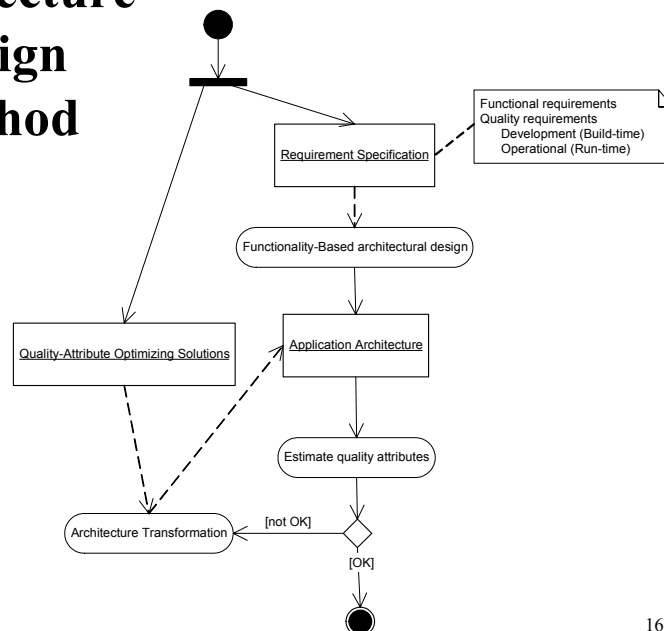
Transformation of Software Architecture

From

Jan Bosch, *Design and Use of Software Architectures: Adopting and evolving a product-line approach*,
ACM Press-Addison Wesley, 2000,
Ch. 6, pp.109-158

167

Architecture Design Method



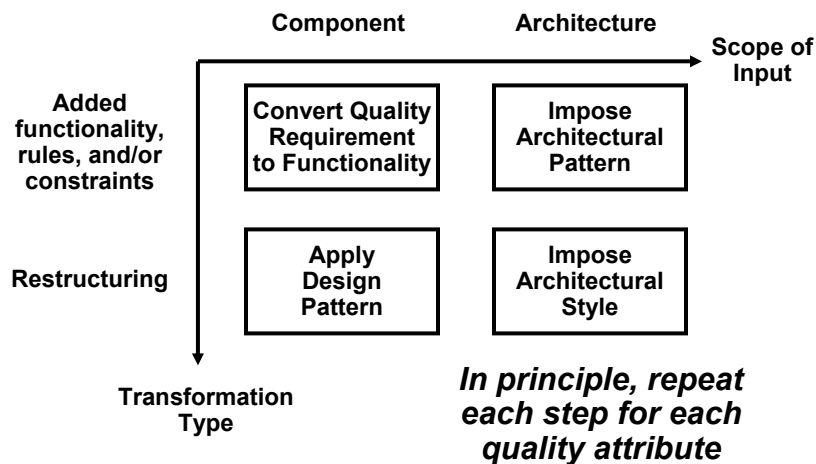
168

Overview of Architecture Design Method

- **Functionality-based architectural design**
 - Define system context
 - Identify archetypes (core functional abstractions)
 - Decompose into components
 - o Interfaces, domains, abstraction layers, domain entities, archetype instantiations
 - Describe system instantiations
- **Assess quality attributes**
 - Define quality profiles
 - Scenario-based assessment
 - Simulation
 - Mathematical modeling
 - Experience-based assessment
- **Architecture transformation**
 - Impose architectural style
 - Impose architectural pattern
 - Impose design patterns
 - Convert quality requirements to functionality
 - Distribute requirements

169

Architecture Transformation Categories



170

Impose an Architectural Style

- Bass *et al.*, Attribute-Based Architectural Styles (ABASs)
 - Pipes and filters
 - Layers
 - Blackboard
 - Object-Orientation
 - Implicit Invocation
 - etc.

171

Impose an Architectural Pattern

- Concurrency
 - Processors, processes, threads, scheduling, etc.
- Persistence
 - Database management system, application-level persistence and transaction handling, etc.
- Distribution
 - Brokers, remote method invocation, etc.
- Graphical User Interface
 - MVC, PAC, etc.

172

Apply a Design Pattern

- Gang-of-Four, Buschmann, etc.
 - Façade
 - Observer
 - Abstract Factory
 - etc.

173

Convert Quality Requirements to Functionality

- Self-monitoring
- Redundancy
- Security
- etc.

Distribute Requirements

- Distribute system-level quality requirements to the subsystems and components
 - System quality X ; component quality x_i
 - $X = x_1 + x_2 + x_3 + \dots + x_n$
- Separate functionally-related qualities
 - e.g., Fault-tolerant computation + fault-tolerant communication

174

Software Product Line Architecture

based on

Jan Bosch

*Design and use of software
architectures*

Addison-Wesley, ACM Press, 2000

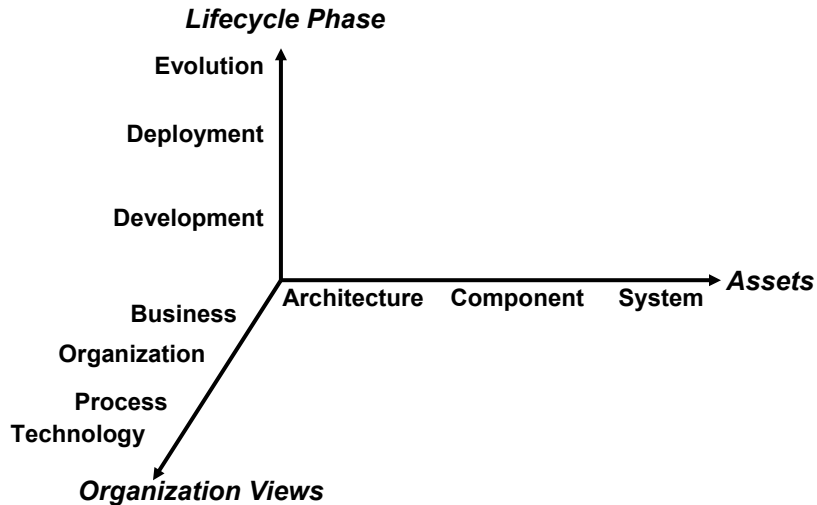
175

Three types of software architecture use

- For an individual software system
- As a product-line architecture
 - common architecture for a set of related products or systems developed by an organization
- As a standard architecture used for a public component market
 - component framework

176

Product line decomposition



177

Assets

- **Architecture**
 - covers all the products in the product line
 - includes features that are shared between products
- **Component**
 - identify components (typically quite large)
 - identify component variability across products
 - develop components
- **System**
 - systems constructed based on product-line architecture and its components
 - add or remove components and component relations
 - develop system-specific component extensions
 - develop system-specific code
 - configure components and system

178

Organization Views

- Business
 - align investment/return roadmap
- Organization
 - organize around domain engineering of architecture and components
- Process
 - define product-line-specific processes
- Technology
 - object-oriented frameworks
 - component assessment with respect to quality attributes (maintainability, etc.)

179

Lifecycle Phase

- Development
 - create initial architecture and components
- Deployment
 - instantiate the product line architecture and components into products (and hence into systems)
 - extend, adapt, configure components
- Evolution
 - evolve assets (architecture, component, and system) to meet changing requirements and technologies

Next, focus on this decomposition of the product line

180

Initiating a Product Line

	Evolutionary	Revolutionary
Existing product line and market	<ul style="list-style-type: none"> • Develop vision based on architectures of existing family of products • Develop one product line component at a time by evolving existing components 	<ul style="list-style-type: none"> • Define new architecture based on superset of product line members' requirements and predicted future requirements
New product line and market	<ul style="list-style-type: none"> • Evolve architecture and components with the evolving requirements posed by new product line members 	<ul style="list-style-type: none"> • Develop product-line architecture and components to match requirements of all expected product-line members

181

Comparison

- Evolve an existing set of products into a product line
 - pro: minimize risk due to lower up-front investment and shorter time-to-market by evolving existing products
 - con: overall time and investment larger
- Replace an existing set of products with a software product line
 - pro: shorter conversion time and smaller overall investment
 - con: increased risk, delayed time to market, existing product line grows stale
- Evolve a new software product line
 - pro: lower up-front investment; shorter time-to-market; evolve domain experience
 - con: requirements of new products may impact architecture, resulting in higher cost
- Develop a new software product line
 - pro: once in place, can develop new products rapidly; lower total investment
 - con: lack of domain experience make it difficult to anticipate requirements, hence architecture and components may not support

182

The Role of a Software Architect

183

Software Architect Role

(from Hofmeister *et al.*, *Applied Software Architecture*)

- The software architect creates a vision
 - keeps up with innovations and technologies
 - understands global requirements and constraints
 - creates a vision (global view) of the system
 - communicates the vision effectively
 - provides requirements and inputs to the system architect
- The software architect is the key technical consultant
 - organizes the development team around the design
 - manages dependencies
 - reviews and negotiates requirements
 - provides inputs regarding technical capabilities of staff
 - motivates the team
 - recommends technology, training, tools
 - tracks the quality of the design
 - ensures architecture meets its design goals

184

Software Architect Role

(continued)

- The software architect makes decisions
 - leads the design team
 - makes early design decisions (key global ones)
 - knows when to end discussion and make a decision
 - identifies and manages risk
- The software architect coaches
 - establishes dialog with each team member
 - teaches the team the design and gets their buy-in
 - listens to feedback
 - knows when to yield to design changes
 - knows when to let others take over detailed design
- The software architect coordinates
 - coordinates activities of tasks that influence or are influenced by the architecture
 - maintains integrity of the design
 - ensures that the architecture is followed

185

Software Architect Role

(continued)

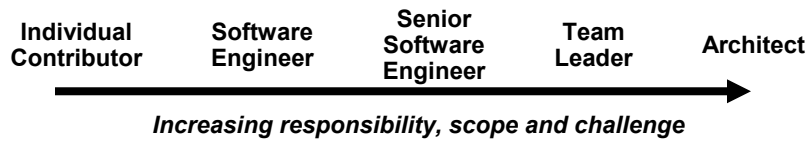
- The software architect implements
 - considers the design implications of introducing a new technology
 - may look at low-level details to validate initial concepts
 - may prototype to explore and evaluate design decisions
 - may implement a thin vertical slice to minimize implementation risk
 - may implement components as an implementation model for developers
- The software architect advocates
 - advocates investment in software architecture
 - works to incorporate software architecture into the software process
 - continues to assess and advocate new software architecture technologies
 - advocates architecture reuse

186

Career Path

(from Hofmeister *et al.*, *Applied Software Architecture*)

- Set your sights on becoming an expert in software engineering
 - gather broad experience
 - develop technical, leadership, communication and people skills
- Apprentice (hang out) with an experienced architect



187

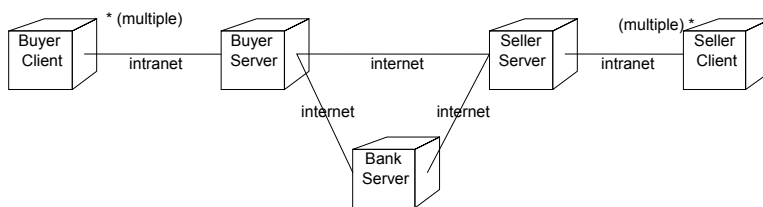
Examples and Details

188

An Interbank System

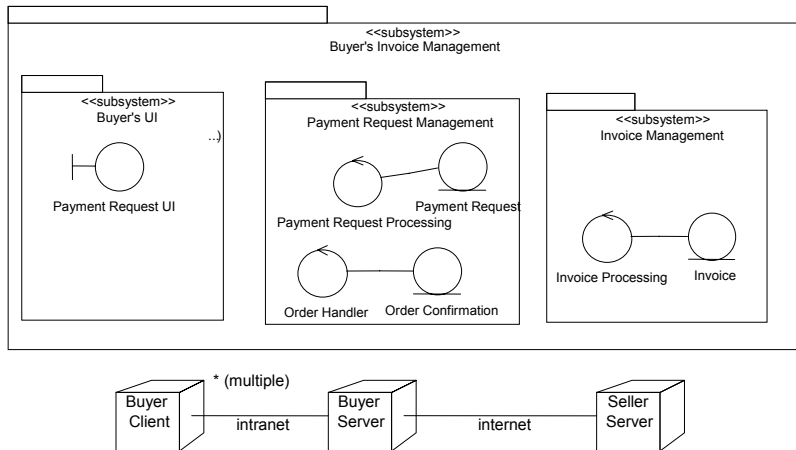
189

UML Deployment Diagram for Interbank System



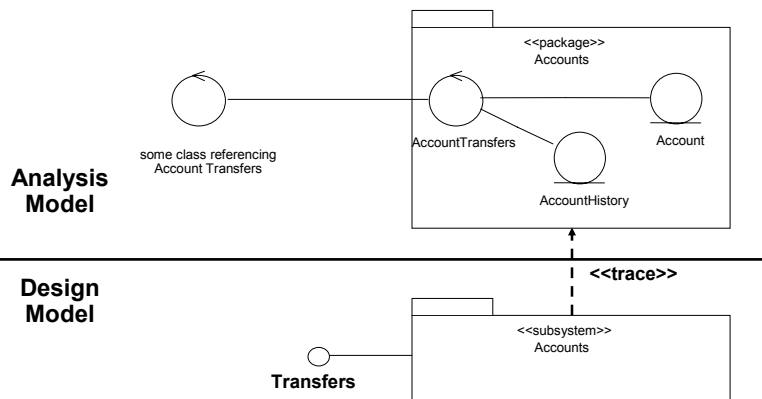
190

Decomposing a Subsystem for Deployment



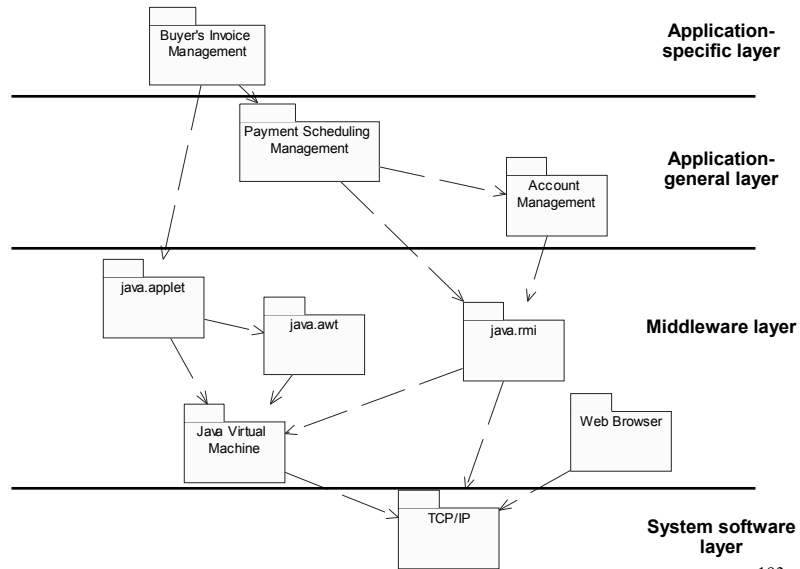
191

Subsystem Interface to Serve Dependencies



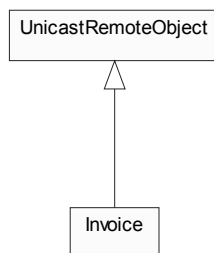
192

Layered architecture: Package dependencies



193

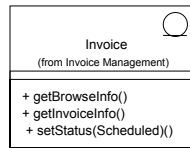
Add a mechanism to handle special requirements



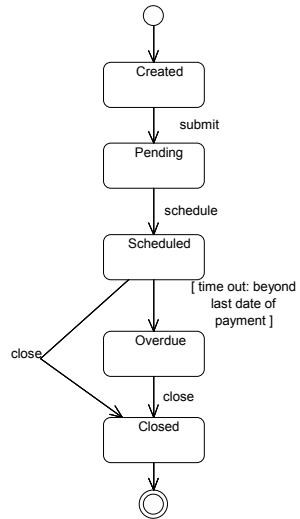
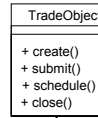
**Invoice objects must be distributed
(accessible across the network)
so make a subclass of
java.rmi.UnicastRemoteObject**

194

Refactoring Example



Refactor



195

A Look at a Repository Style

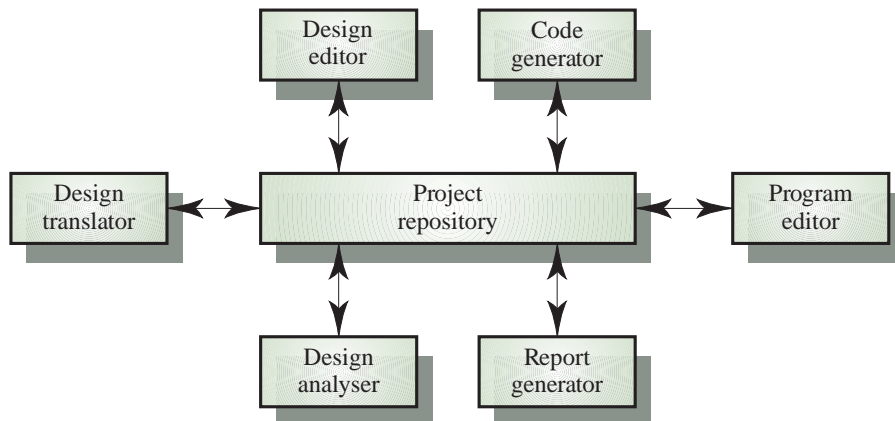
196

The repository model

- Sub-systems must exchange data. This may be done in two ways:
 - Shared data is held in a central database or repository and may be accessed by all sub-systems
 - Each sub-system maintains its own database and passes data explicitly to other sub-systems
- When large amounts of data are to be shared, the repository model of sharing is most commonly used

197

CASE toolset architecture



198

Repository Model Characteristics

- **Advantages**
 - Efficient way to share large amounts of data
 - Sub-systems need not be concerned with how data is produced Centralised management e.g. backup, security, etc.
 - Sharing model is published as the repository schema
- **Disadvantages**
 - Sub-systems must agree on a repository data model. Inevitably a compromise
 - Data evolution is difficult and expensive
 - No scope for specific management policies
 - Difficult to distribute efficiently

199

A Look at Client-Server

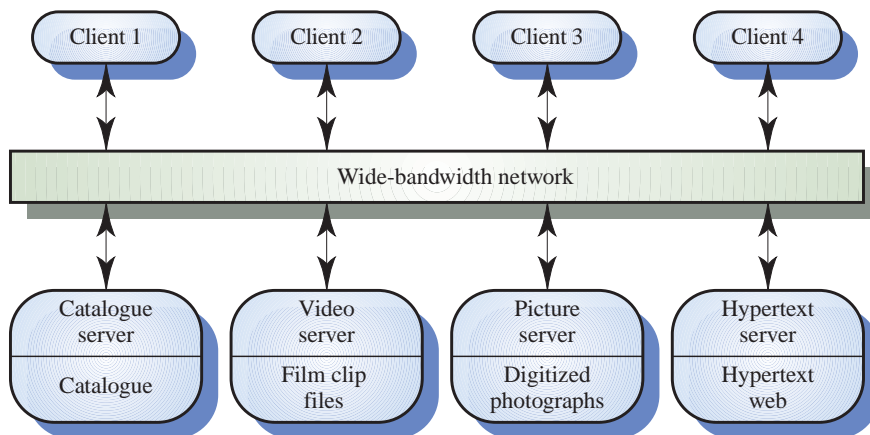
200

Client-Server Architecture

- Distributed system model which shows how data and processing is distributed across a range of components
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services
- Network which allows clients to access servers

201

Film and Picture Library



202

Client-server characteristics

- **Advantages**
 - Distribution of data is straightforward
 - Makes effective use of networked systems. May allow cheaper hardware
 - Easy to add new servers or upgrade existing servers
- **Disadvantages**
 - No shared data model so sub-systems use different data organisation. Data interchange may be inefficient
 - Redundant management in each server
 - No central register of names and services - it may be hard to find out what servers and services are available

203

A Look at Some Control Models

204

Control Models

- Are concerned with the control flow between sub-systems. Distinct from the system decomposition model
- Centralized control
 - One sub-system has overall responsibility for control and starts and stops other sub-systems
- Event-based control
 - Each sub-system can respond to externally generated events from other sub-systems or the system's environment

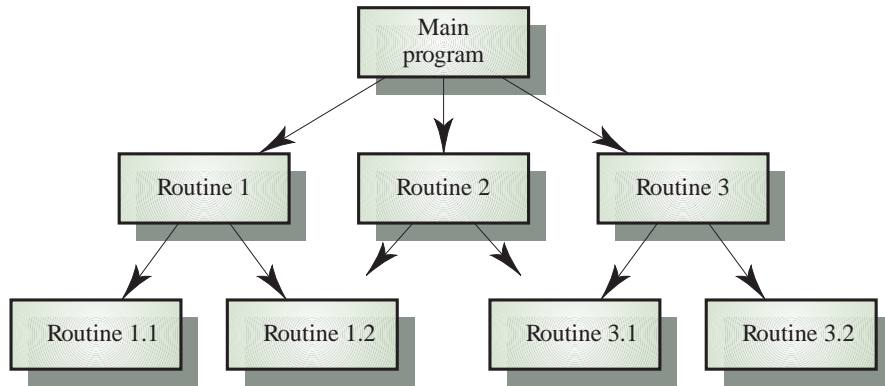
205

Centralized Control

- A control sub-system takes responsibility for managing the execution of other sub-systems
- Call-return model
 - Top-down subroutine model where control starts at the top of a subroutine hierarchy and moves downwards. Applicable to sequential systems
- Manager model
 - Applicable to concurrent systems. One system component controls the stopping, starting and coordination of other system processes. Can be implemented in sequential systems as a case statement

206

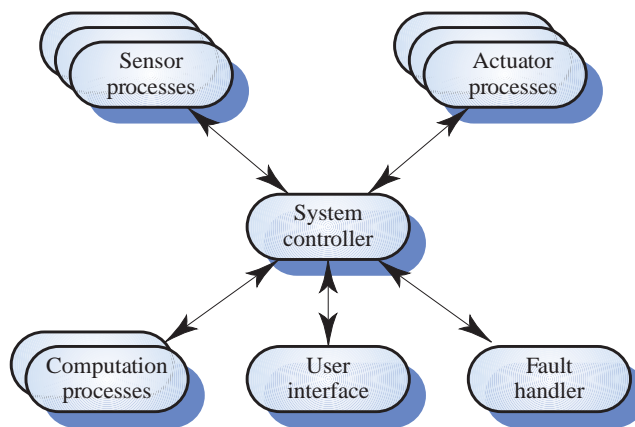
Call-Return Model



207

Real-time system control

A Manager Model



208

Event-driven systems

- Driven by externally generated events where the timing of the event is outside the control of the sub-systems which process the event
- Two principal event-driven models
 - Broadcast models. An event is broadcast to all sub-systems. Any sub-system which can handle the event may do so
 - Interrupt-driven models. Used in real-time systems where interrupts are detected by an interrupt handler and passed to some other component for processing

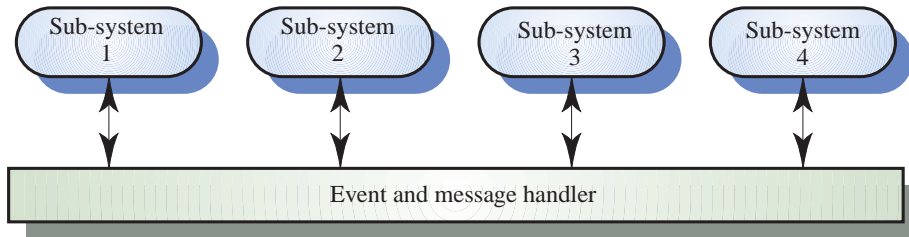
209

Broadcast model

- Effective in integrating sub-systems on different computers in a network
- Sub-systems register an interest in specific events. When these occur, control is transferred to the sub-system which can handle the event
 - Publish/Subscribe
- Control policy is not embedded in the event and message handler. Sub-systems decide on events of interest to them
- However, sub-systems don't know if or when an event will be handled

210

Selective Broadcasting



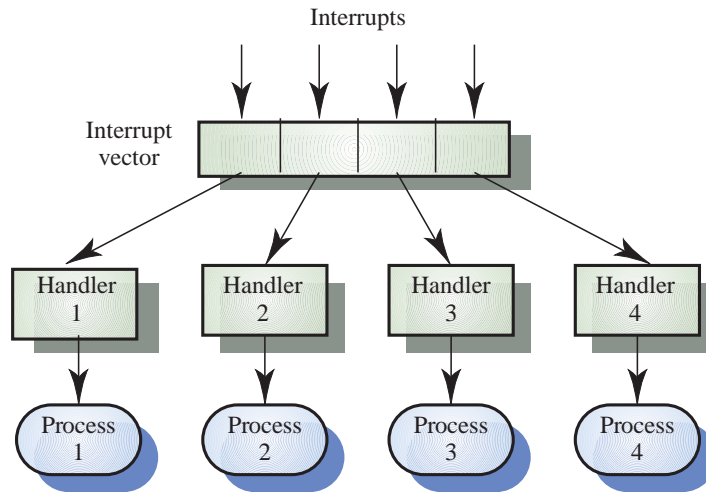
211

Interrupt-Driven Systems

- Used in real-time systems where fast response to an event is essential
- There are known interrupt types with a handler defined for each type
- Each type is associated with a memory location and a hardware switch causes transfer to its handler
- Allows fast response but complex to program and difficult to validate

212

Interrupt-Driven Control



213

Distributed System Architectures

214

Distributed System Characteristics

- Resource sharing
- Openness
- Concurrency
- Scalability
- Fault tolerance
- Transparency

215

Distributed System Disadvantages

- Complexity
- Security
- Manageability
- Unpredictability

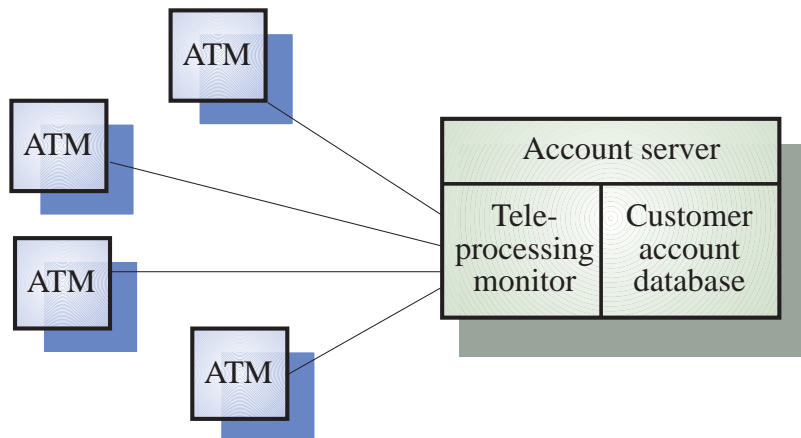
216

Multiprocessor architectures

- Simplest distributed system model
- System composed of multiple processes which may (but need not) execute on different processors
- Architectural model of many large real-time systems
- Distribution of process to processor may be pre-ordered or may be under the control of a dispatcher

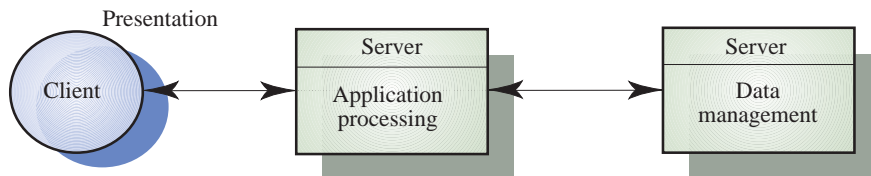
217

A Client-Server ATM System



218

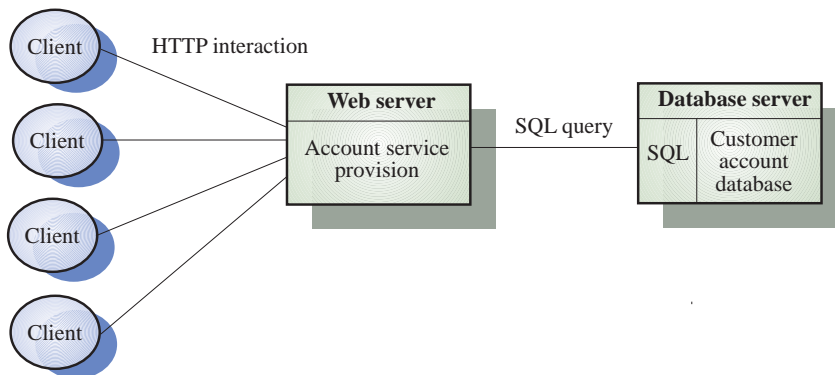
A Distributed Three-Tier Client-Server Architecture



Replication at any tier enables scaling and reliability

219

An Internet Banking System



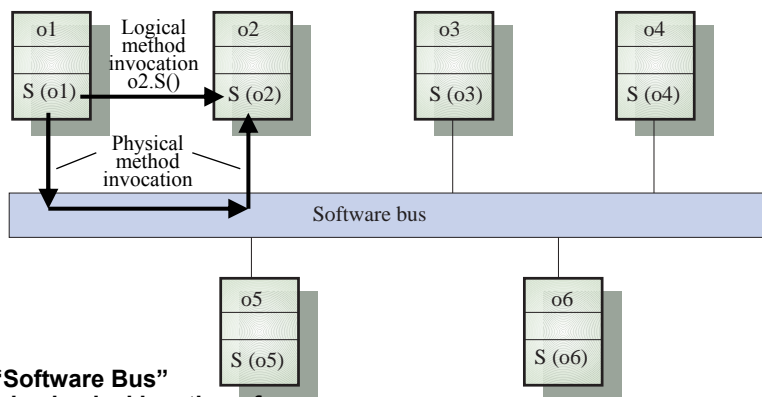
220

Distributed Object Architectures

- There is no distinction in a distributed object architecture between clients and servers
 - Peer-to-peer
- Each distributable entity is an object that provides services to other objects and receives services from other objects
 - Each object can be client and/or server of services
- Object communication is through a middleware system called an object request broker
 - Software bus
- However, more complex to design than client-server systems

221

Distributed Object Architecture



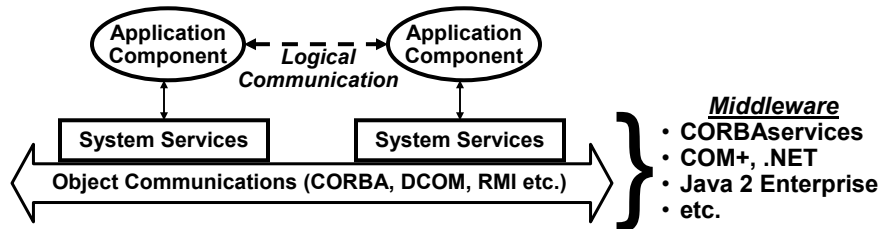
The "Software Bus"

- Finds physical location of servers
- Manages conversion between programming languages, data representations, and operating systems
- Manages the mechanics of invoking methods and returning results

222

Middleware

- Software that manages and supports the different components of a distributed system. In essence, it sits in the *middle* of the system
- Middleware is usually off-the-shelf rather than specially written software
- Examples
 - Transaction processing monitors
 - Data converters
 - Communication controllers



223

Middleware

- Software that manages and supports the different components of a distributed system. In essence, it sits in the *middle* of the system
- Middleware is usually off-the-shelf rather than specially written software
- Examples
 - Transaction processing monitors
 - Data converters
 - Communication controllers
 - Object Request Brokers (Software bus)
 - Messaging and Event Brokers

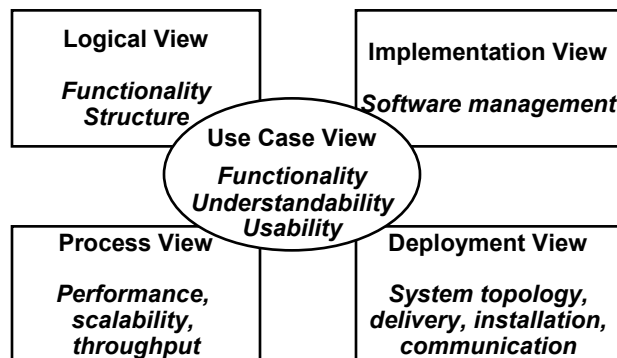
224

Describe the Distribution Architecture

From the Rational Unified Process

225

Focusing on the Deployment View



The Deployment View is an architecturally significant slice of the Deployment Model

- Physical nodes
- Interconnections
- Allocations of threads of control to the physical nodes

226

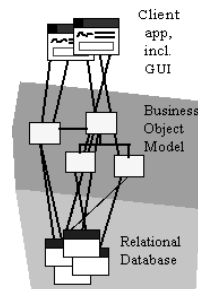
Why Distribute?

- Reduce processor overload
- Special processing requirements
 - e.g., special-purpose hardware, such as signal processing or communications
- Scalability concerns
- Economic concerns
 - cheaper to have multiple small processors than one large processor
- Access to the system distributed across multiple locations
- Caution: Often the overhead of interprocessor communication negates the benefit of distribution
 - Carefully understand the patterns of communication

227

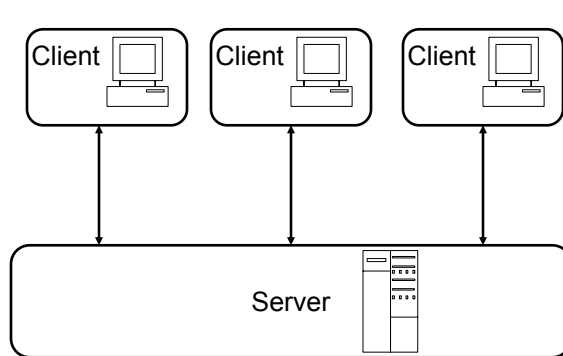
Distribution Patterns

- Client/Server (client nodes consume services of server nodes)
 - 3-tier
 - Functionality equally divided across three partitions: application, business, and data services
 - Fat client
 - Place functionality in client (such as, databases or file servers)
 - Fat server (thin client)
 - Place more functionality in the server (such as, groupware, transaction servers, web servers)
 - Distributed client/server
 - Business services and data services further distributed across nodes
- Peer-to-peer (any process or node may be a client and a server)



228

Client/Server Architecture



• Client

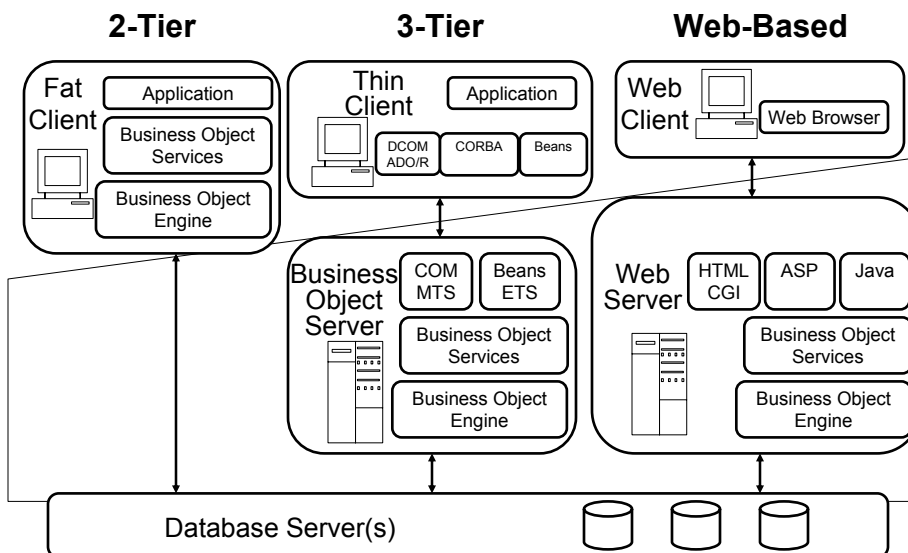
- Typically services a single user
- Provides user interface services

• Server

- Services multiple clients simultaneously
- Provides database, security, print, file, communications, and other services

229

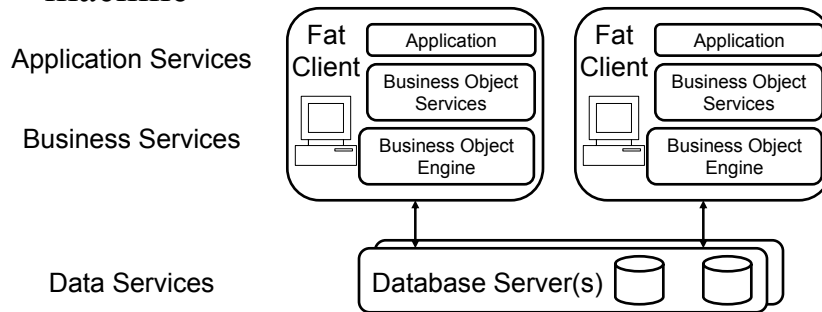
Client/Server Architectures



230

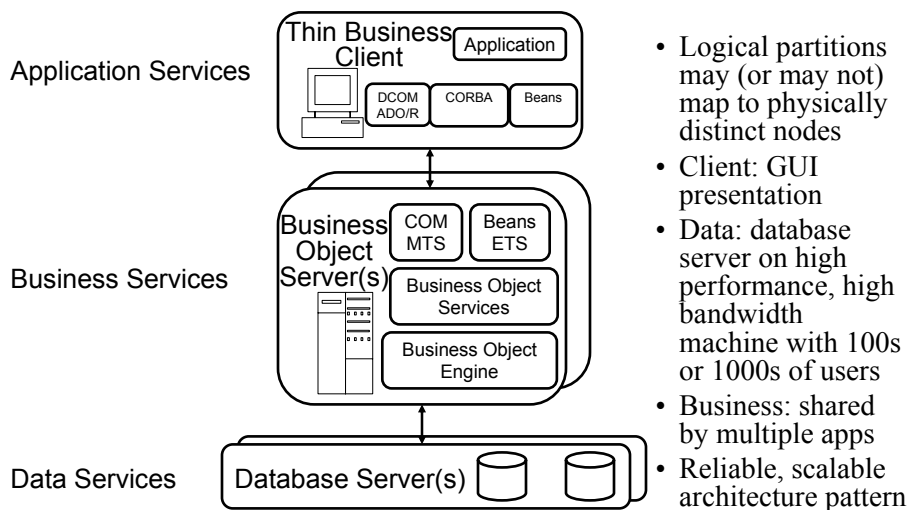
Client/Server: “Fat Client” 2-Tier Architecture

- Database server typically on a separate machine



231

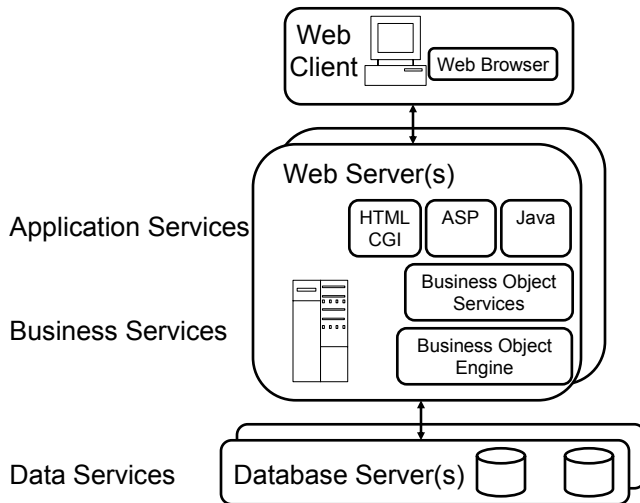
Three-Tier Client/Server



- Logical partitions may (or may not) map to physically distinct nodes
- Client: GUI presentation
- Data: database server on high performance, high bandwidth machine with 100s or 1000s of users
- Business: shared by multiple apps
- Reliable, scalable architecture pattern

232

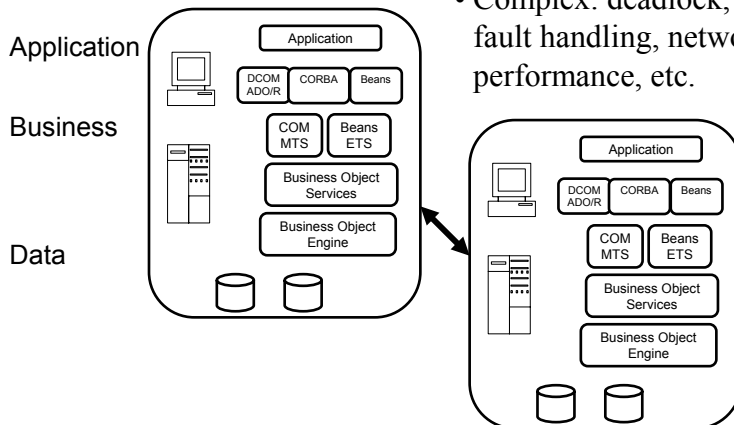
Client/Server: Web Application Architecture



- Client is a web browser running HTML pages, Java applets, Java Beans, or ActiveX components
- Easy to distribute and change

233

Peer-to-Peer Architecture



- Group related services together to minimize network traffic while maximizing throughput and system utilization
- Complex: deadlock, starvation, fault handling, network performance, etc.

234

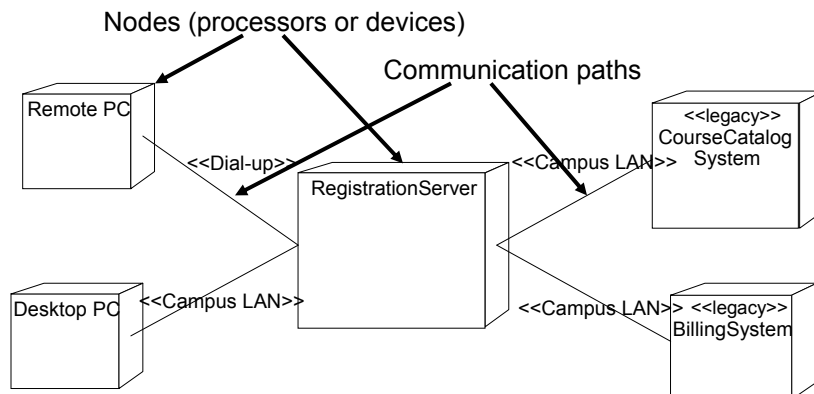
Describe the Distribution: Steps

- Define the Network Configuration
- Allocate Processes to Nodes
- Define Distribution Mechanisms

235

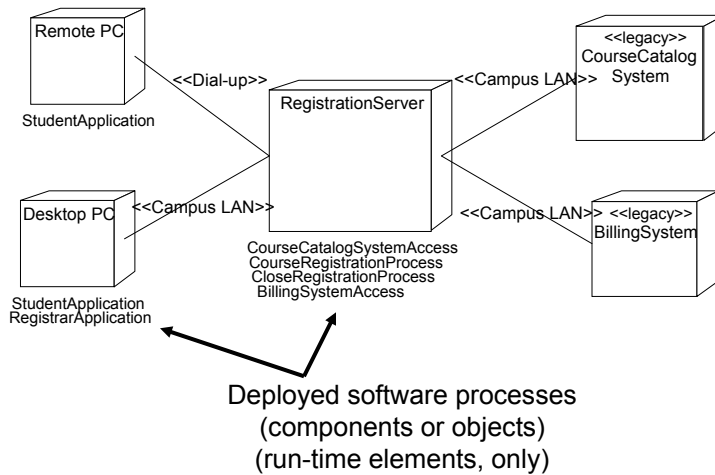
Example: Network Configuration

UML Deployment Diagram



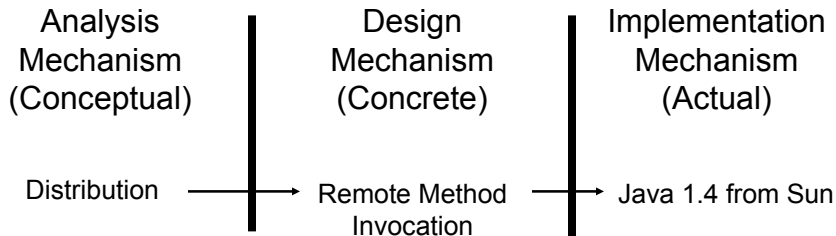
236

Allocate Processes to Nodes



237

Distribution Mechanism



- Options for Distributed Objects
 - Microsoft DCOM and .NET (Microsoft platforms only)
 - Java Remote Method Invocation (Java only)
 - CORBA (Multi-language, multi-platform, robust, heavyweight)
- Non-object options
 - Message-oriented Middleware (TIBCO, MMQ, etc.)
 - etc.

238

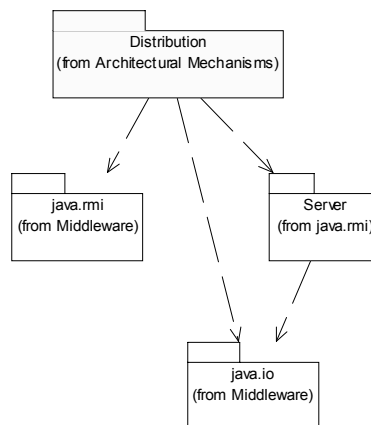
Design Mechanism: Distribution: RMI

- Distribution characteristics
 - Latency
 - Synchronous or asynchronous
 - Message size
 - Protocol

239

Remote Method Invocation (RMI)

- Java library packages used

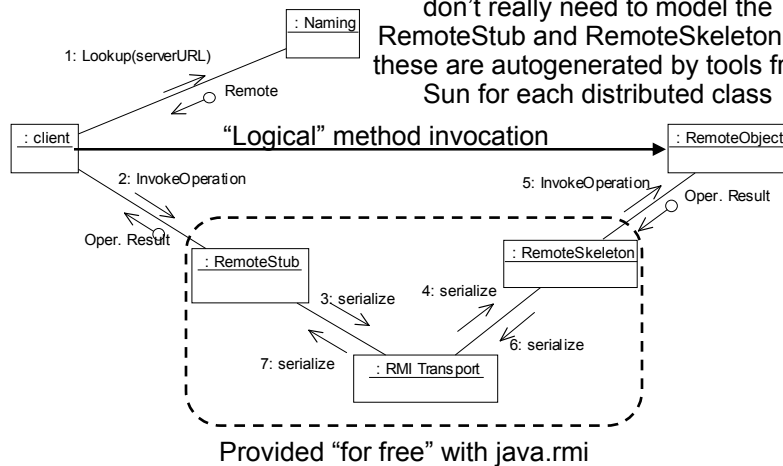


240

RMI Operation

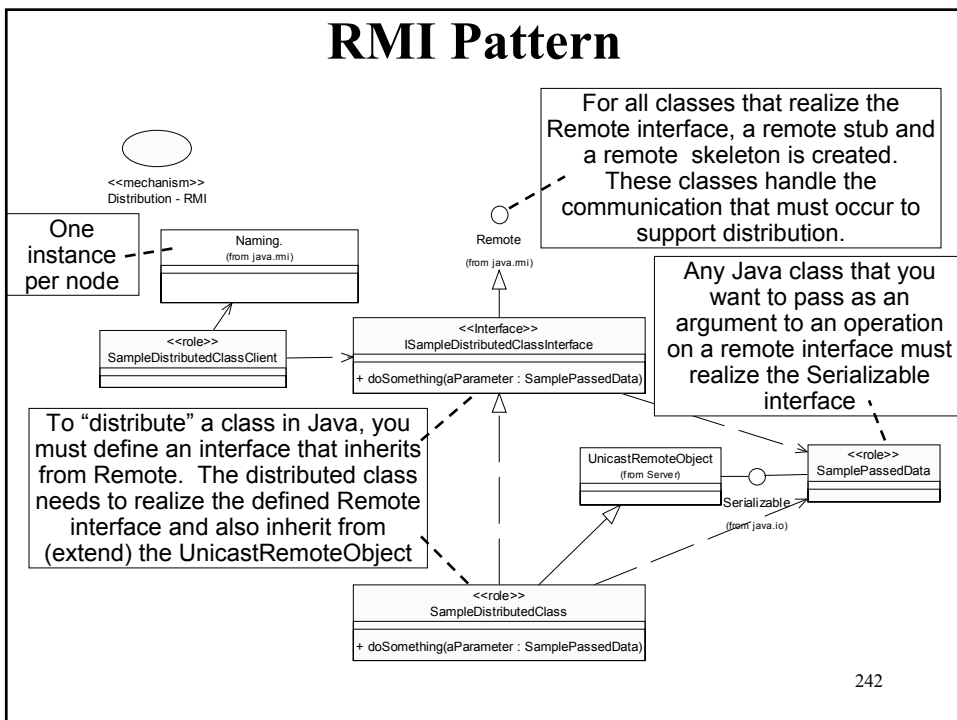
(continued)

This diagram describes what happens “under the hood”, but in reality, you don’t really need to model the RemoteStub and RemoteSkeleton as these are autogenerated by tools from Sun for each distributed class



241

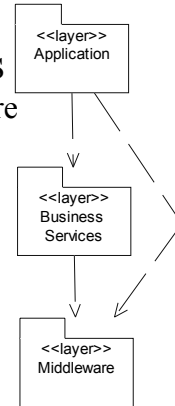
RMI Pattern



242

Incorporating RMI: Steps

- Provide access to RMI support classes
 - Use java.rmi and java.io package in Middleware layer
- For each class to be distributed:
 - Put control classes to be distributed in the Application layer
 - Need dependency from Application layer to Middleware layer to access java packages
 - Define interface for class that realizes Remote
 - In detailed design
 - Remote has no operations to implement
 - Have class inherit from UnicastRemoteObject



(continued)

243

Incorporating RMI: Steps

(continued)

- Have classes for data passed to distributed objects realize the Serializable interface
 - Core data types are in Business Services layer
 - Need dependency from Business Services layer to Middleware layer to get access to java.rmi
 - Add the realization relationships
 - No operations to implement
- Run rmic pre-processor to generate the stubs and skeletons for all classes that realize the Remote interface
 - Include stub as a local class in client
 - Local proxy for the remote server
 - Include skeleton as a local class in server
 - Local proxy for the remote client

(continued)

244

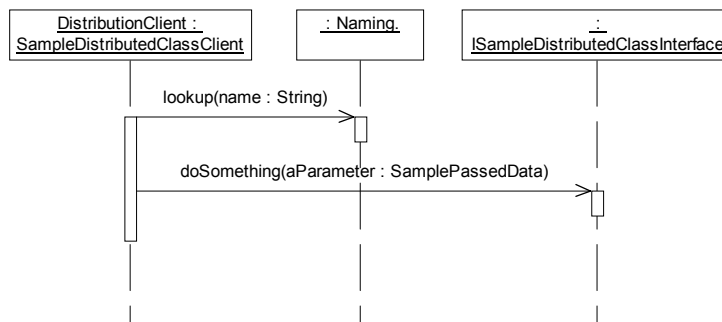
Incorporating RMI: Steps

(continued)

- Have distributed class clients lookup the remote objects using the Naming service
 - Most distributed class clients are Forms
 - Forms are in the Application layer
 - Need dependency from Application layer to Middleware layer to get access to java.rmi
 - Add relationship from Distributed Class Clients to Naming Service
- Create/update interaction diagrams with distribution processing
 - Just model the Naming service and the distributed class interface
 - Don't model the Remote Stub and Skeleton

245

Example Interaction Diagram



246

Basic Distributed Client/Server Architecture

The diagram illustrates the Basic Distributed Client/Server Architecture, showing the flow of data and components between the Client and Server sides.

Client Side Components:

- Client Application:** The top-level application on the client side.
- Middle-ware:** The layer between the Client Application and the Client Platform.
- Client Platform:** The hardware and operating system layer on the client side.

Server Side Components:

- Server Program:** The top-level application on the server side.
- Middle-ware:** The layer between the Server Program and the Server Platform.
- Server Platform:** The hardware and operating system layer on the server side.

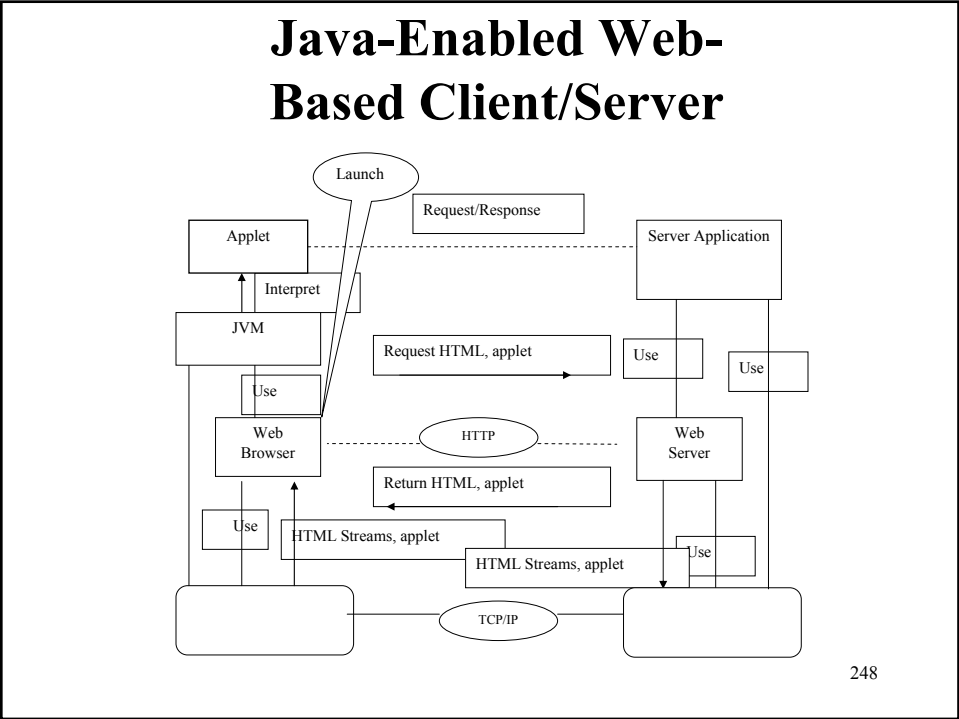
Communication and Protocols:

- Application Protocol:** Service Request/Response (indicated by a dashed line between Client Application and Server Program).
- Distributed Client/Server Protocol:** (indicated by a dashed line between Client Middle-ware and Server Middle-ware).
- Network Protocol:** (indicated by a dashed line between Client Platform and Server Platform).
- Physical Communication:** (indicated by a solid line between Client Platform and Server Platform).

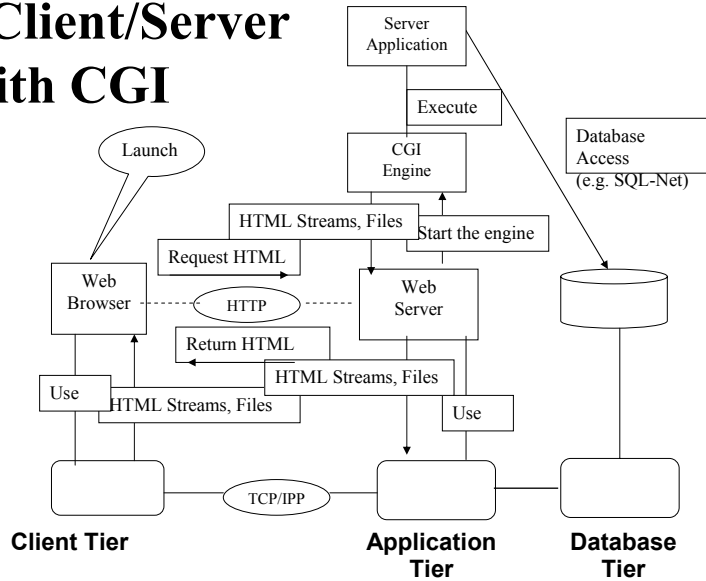
Numbered Data Flows:

- Client Application to Middle-ware
- Middle-ware to Client Platform
- Client Platform to Physical Communication
- Physical Communication to Server Platform
- Server Platform to Middle-ware
- Middle-ware to Server Program
- Server Program to Middle-ware
- Middle-ware to Server Platform
- Server Platform to Middle-ware
- Server Platform to Physical Communication
- Physical Communication to Client Platform
- Client Platform to Middle-ware
- Middle-ware to Client Application
- Client Application to Middle-ware

247

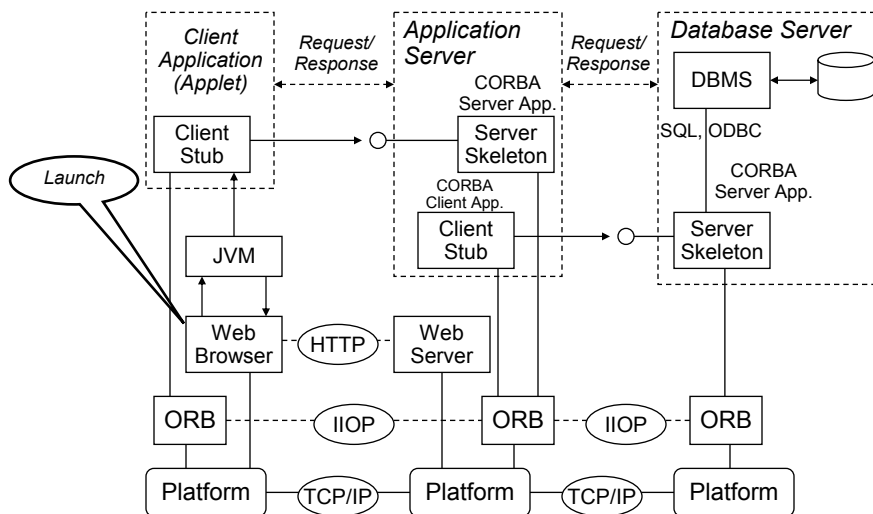


Three-Tier, Web-Based Client/Server with CGI



249

Web, CORBA, Three-Tier, Client/Server



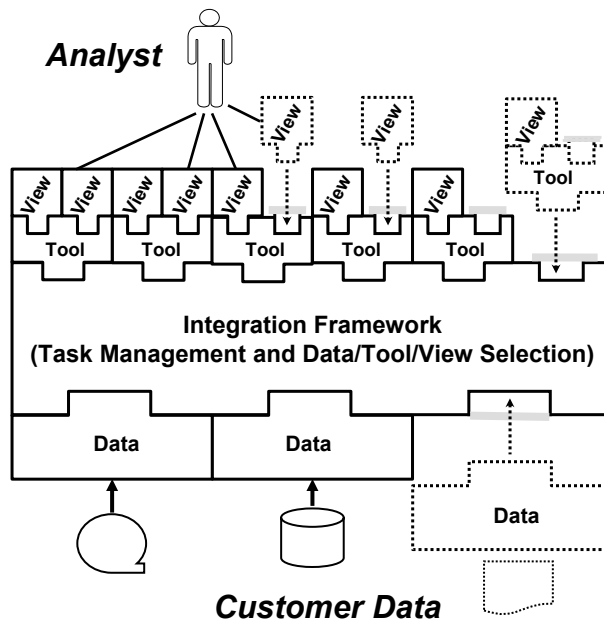
250

A Sample Architecture for a Crash Analysis and Reporting Environment

251

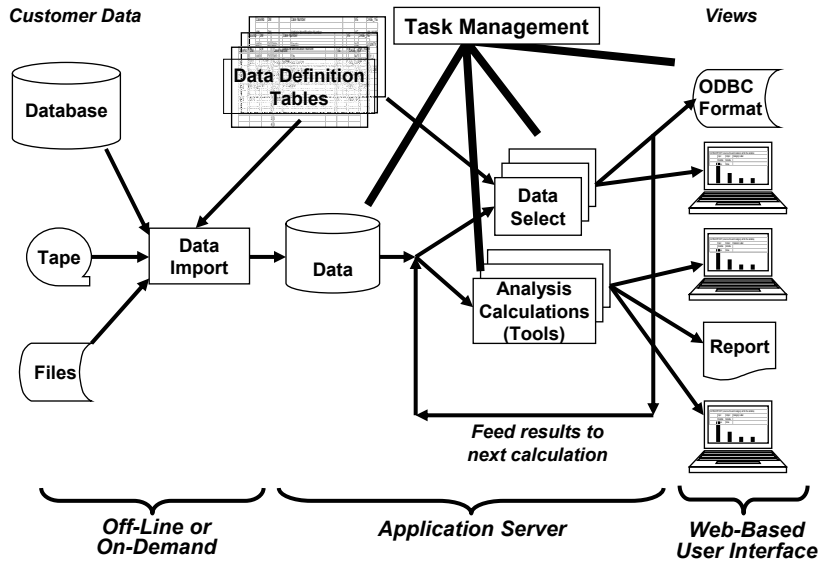
Plug-In Architecture

- Problem:
Difficult to add
new analysis
tools ,
visualizations,
and data sets to
a large-scale
data analysis
and information
mining system
- Solution: A
framework to
host and
integrate plug-in
tools and data



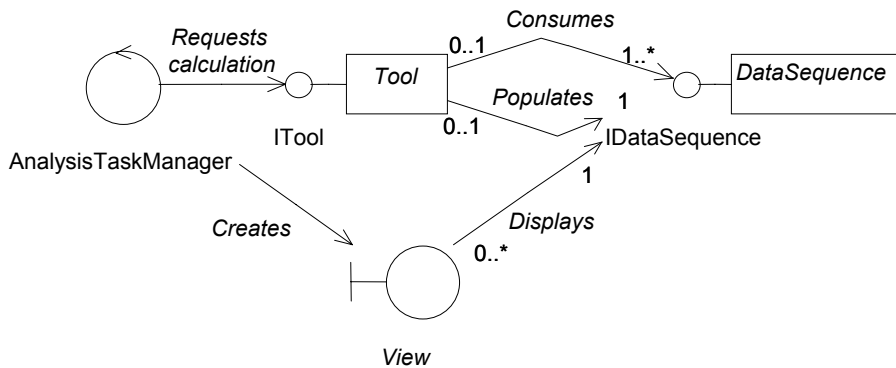
252

Block Diagram



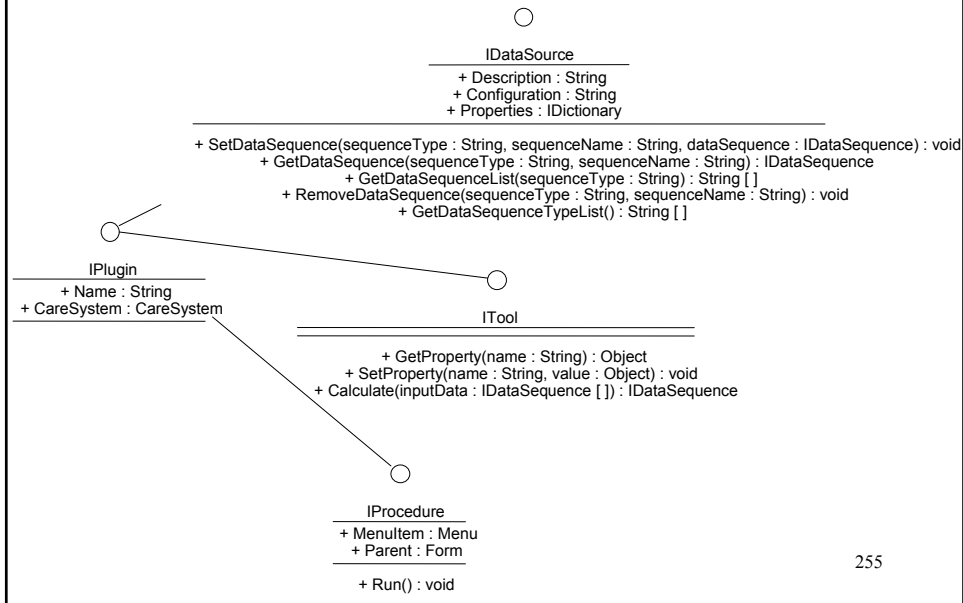
253

Tool, View, and Data Plug-in Framework Architecture



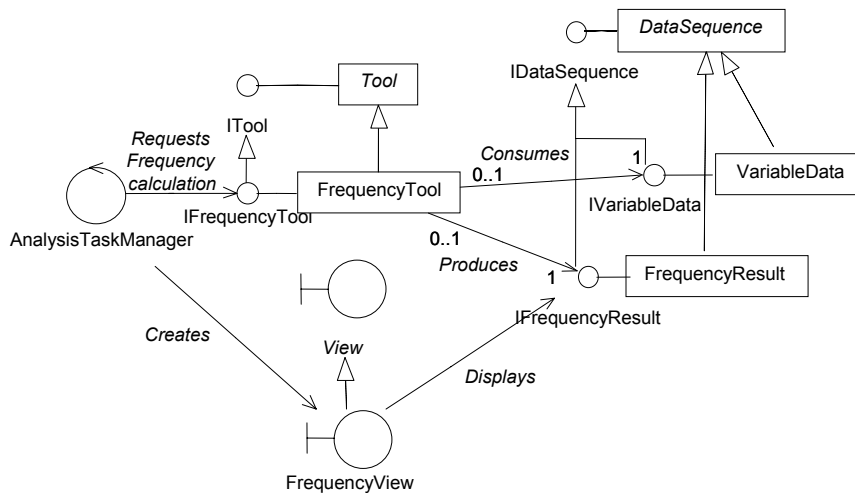
254

Interface Specifications for Plug-ins



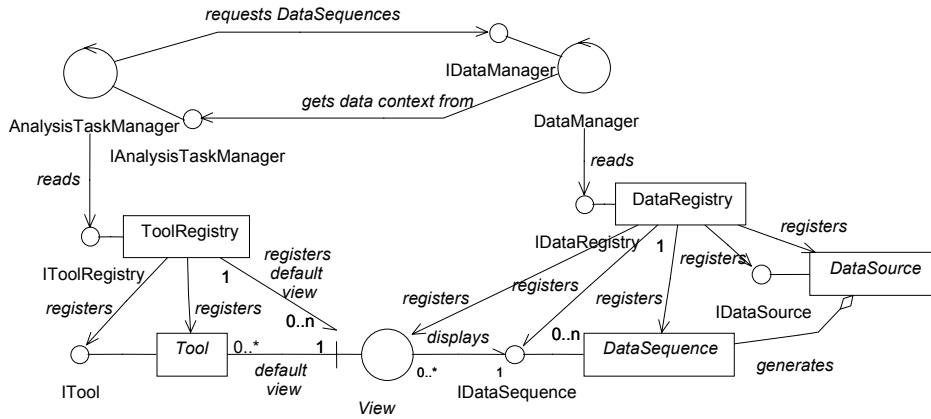
255

Frequency Calculation Specialization of the Plug-in Framework



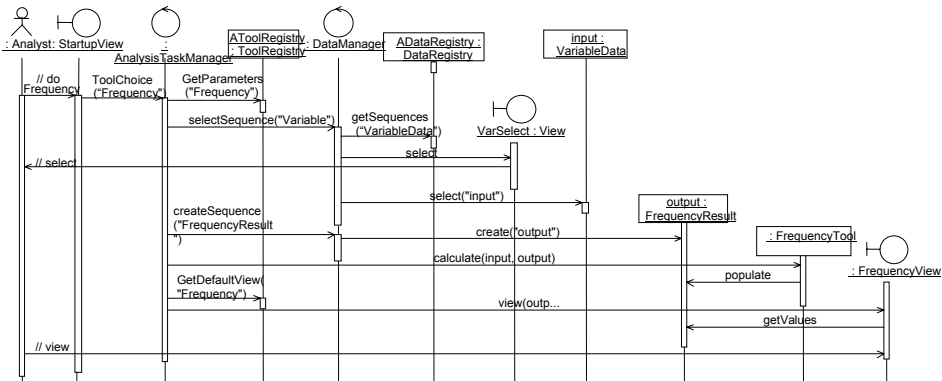
256

Registries



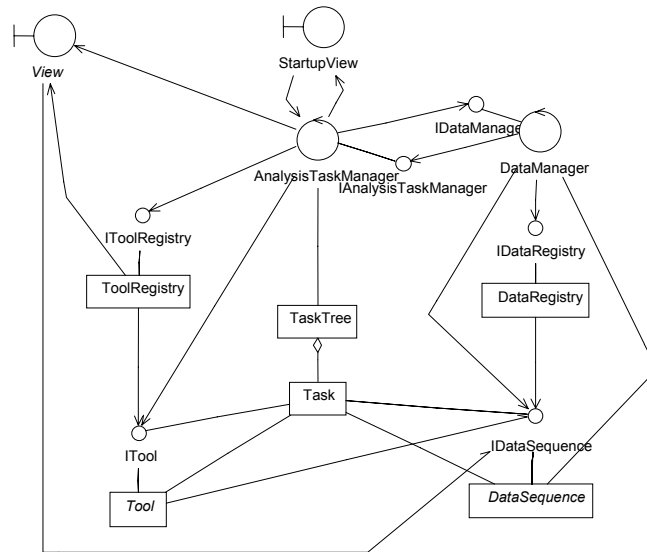
257

Frequency Operation



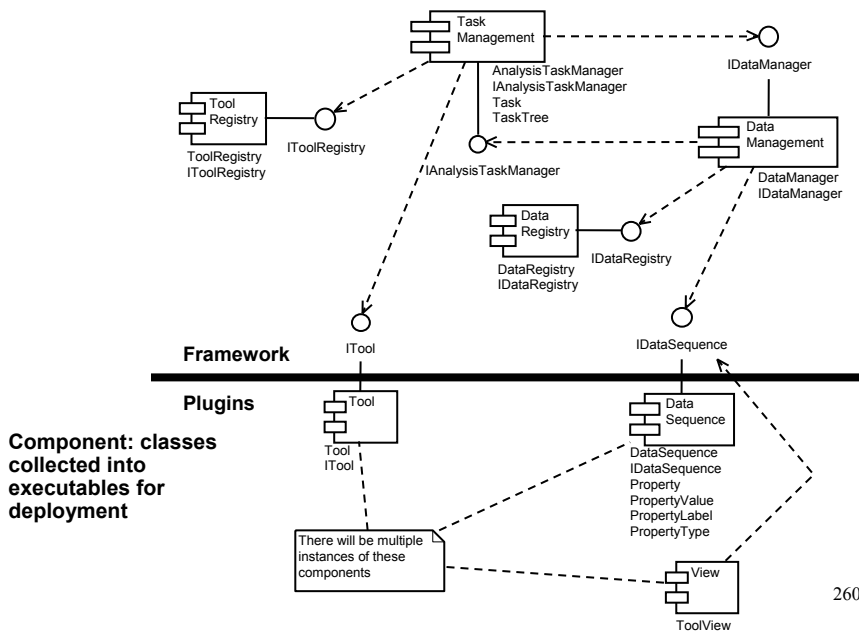
258

Plug-in Architecture Summary



259

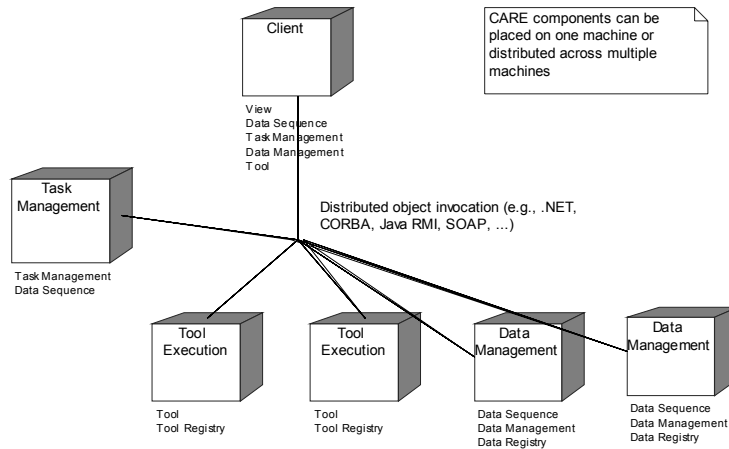
Component Architecture



Component: classes collected into executables for deployment

260

Deployment Architecture



261

World Wide Web Architecture Case Study


262

Initial WWW

- 1989: Support “human web” of collaborative researchers in dynamic European Laboratory for Particle Physics
 - “Information Management: A Proposal”
 - High risk, High payoff, Low investment
 - Authors and readers create links and annotate information

263

Requirements Overview

- Portable across platform types
 - Interoperable with implementations on other platform types
 - Scalable
 - Extensible
 - Promote human interaction
 - remote access
 - interoperable
 - extensible
 - scalable
 - Strict separation of concerns
 - new protocols
 - new data formats
 - new applications
- 
- libWWW library for**
- **web-based development**
 - **distributed client-server architecture**

264

Original Requirements and Qualities

- Remote access across networks
- Heterogeneous computing environment
- No centralized control (no central info server or link creation)
- Access to existing data
- ASCII 24x80 character terminal
- Users can add data
 - Using same interface used to read data – reader annotation and publish
- Permit data analysis (not done)
- Private link annotations (not done)
- Live links to new data
 - re-retrieve information with each access
 - notify link user that information has changed (not done)

265

Original *Non*-Requirements

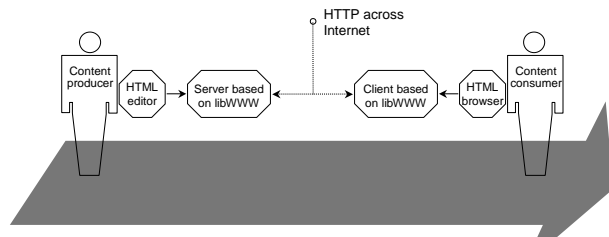
- Controlling the web topology
- Navigational techniques
- User interface requirements (now dominant)
- Graphics and multimedia (now dominant)
- Different link types
- Visual history
- Copyright enforcement
- Security (now important)
- Privacy
- Mark-up format

Notice the changing requirements

266

Architectural Approach

- Scalable: client/Server; references to other data local to location of referring data
- Remote: WWW on internet
- Interop: libWWW masks platform
- Extensible software: protocol and data types isolated in libWWW
- Extensible data: each data item is independent except for its references
- Note separation of UI



267

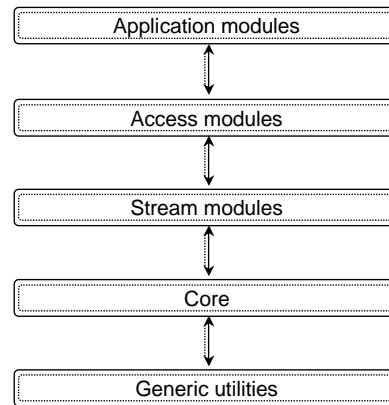
How WWW Architecture Achieves Qualities

<u>Goal</u>	<u>How achieved</u>
Remote access	Build on top of internet
Interoperability	libWWW masks platform details
Extensibility of software	Isolate protocol and data type extensions in libWWW
Extensibility of data	Each data item is independent except for references it controls
Scalability	Use client-server architecture; Keep references to other data local to the referring data location

268

Meeting the Requirements: libWWW

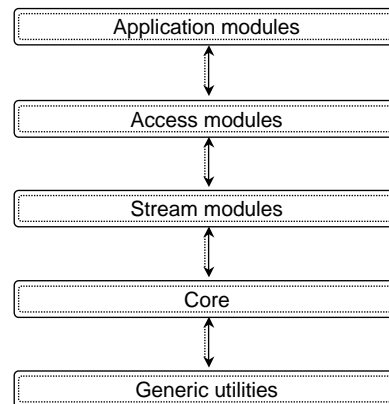
- Compact, portable library that could be built on to create clients, servers, databases, spiders, etc.
- Common functionality: connect, understand HTML streams, etc.
- Run-time concurrency
view of architecture is client-server
- Development view is a layered architecture



269

Generic Utilities Layer

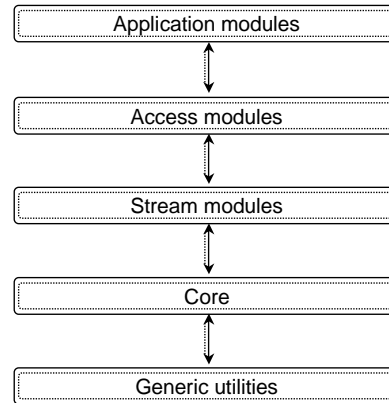
- Portability layer
- Basic building blocks for the system
 - network management
 - data types (such as container classes)
 - string manipulation utilities



270

Core Layer

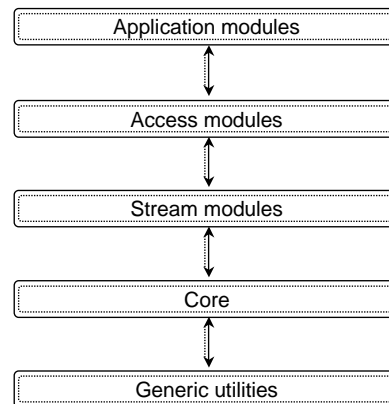
- Skeletal functionality of a WWW application
 - network access
 - data management
 - parsing
 - logging
- Core, by itself, does nothing (it is a standard interface)
 - Actual work is done by plug-ins and call-outs registered by the application
 - Protocols, data formats, dynamic additions, pre-/post-processing, etc.
 - Note the extensibility



271

Stream Modules Layer

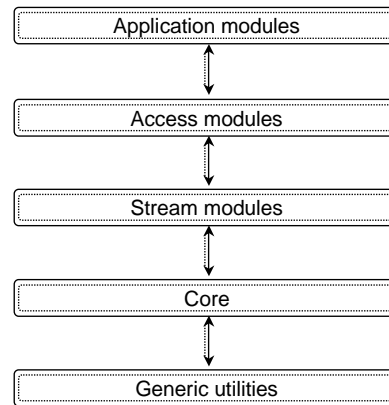
- Abstraction of a stream of data
 - all data transported between the network and the application use streams



272

Access Modules Layer

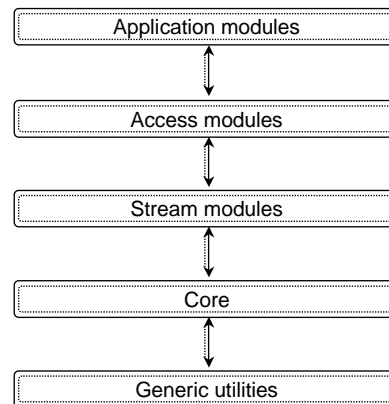
- Skeleton for network protocol-aware modules
 - HTTP
 - NNTP
 - WAIS
 - FTP
 - Telnet
 - rlogin
 - Gopher
 - local file system
 - TN3270
- Easy to add new protocols



273

Application Modules Layer

- Skeleton functionality for building web apps
 - caching
 - logging
 - registering proxy servers (protocol translation)
 - gateways (security firewalls, etc.)
 - history maintenance



274

libWWW

Lessons Learned

- Most challenging requirement was supplying/allowing growth of features – bells and whistles
- Lessons
 - Require formalized, language-independent APIs to libWWW services
 - Layered functionality and APIs
 - APIs must support a dynamic, open-ended set of features
 - including run-time replacement of features
 - APIs must be thread safe to allow concurrent functionality (such as background file download)
- Today: essential, non-replaceable services; pseudo-threads; little dynamic feature replacement

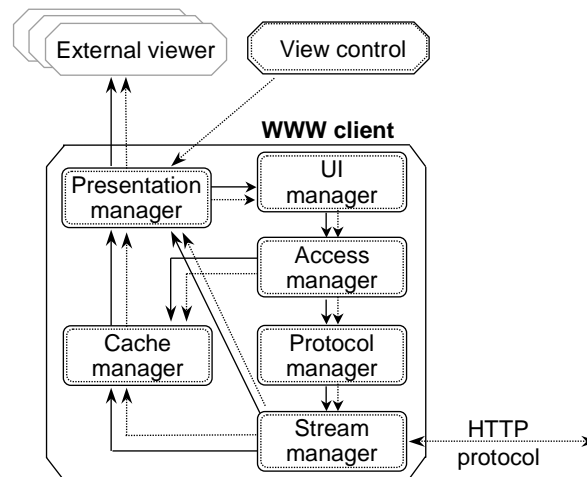
275

Run-Time WWW Client Server Example

- Notes
 - Layers of libWWW disappear in the run-time view
 - Not all functions are built on libWWW
 - e.g., UI functionality is independent of libWWW
 - Names of managers do not correspond to names of layers

276

Typical WWW Client



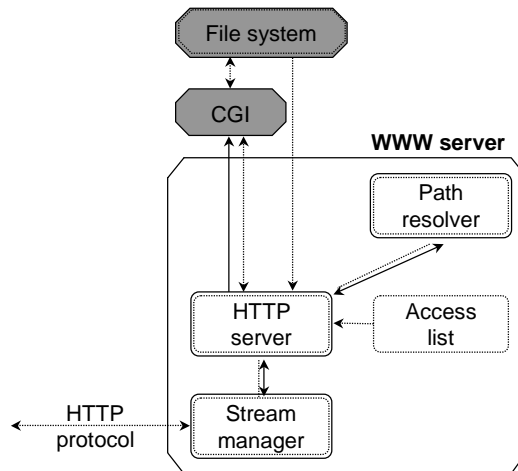
277

Client Components

- User Interface manager is responsible for the look and feel of the UI (e.g., web browser)
- Presentation manager delegates document types to viewers
 - external: QuickTime movies, MP3 audio, etc.
 - internal: HTML, GIFs to UI manager
- UI manager captures user's requests for URL and passes to Access manager
- Access manager determines if URL has been cached; if not, it initiates retrieval through Protocol and Stream managers
- Response stream sent to presentation manager for appropriate display

278

Typical WWW Server



279

Server Components

- Hypertext Transfer Protocol (HTTP) server receives URL request and passes that the Path resolver, which determines the file location for the document (assuming local)
- HTTP server checks the access list to see if access is permitted (may initiate password authentication session) and then gets the document from the file system and writes it to the output stream
- Common Gateway Interface (CGI) is a special document type that allows customized access to other data or programs
 - Also writes to output stream
- HTML stream is served by the HTTP server back to the client

280

Extending the Server

- CGI allows
 - dynamic documents
 - addition of data to existing databases
 - customized queries
 - clickable images
- CGI scripts
 - can be written in a variety of languages
 - run as separate processes

281

Problems with CGI

- Extensibility and “user put” principally supported by CGI scripts in libWWW applications
 - Only partially supports reader publication of annotations, information, etc.
- CGI security holes
- CGI is not portable
- Limits future growth of libWWW-based applications
 - Consider Jigsaw: object-oriented server written in Java

282

Why Did the WWW Succeed?

- Recall that the original concept of the WWW allowed the reader of data to add data and re-publish the result (e.g., annotate a document and re-publish it with the annotations).
- How might this have affected the success of the WWW?
- What are architectural implications of adding this requirement to the original libWWW-based architecture?

283

Attribute-Based Architecture Styles

Reference

Mark Klein, Rick Kazman, “Attribute-Based Architecture Styles” Technical Report CMU/SEI-99-TR-022, Software Engineering Institute, October 1999.

284

Architecture Styles

- **Architecture Style:** classes of architectural designs with their associated known properties
 - description of component types, topology and description of pattern of data and control interaction among components
 - informal description of benefits and drawbacks of using that style
 - experience-based evidence of how each style has been used historically, along with qualitative reasoning to explain why each style has its specific properties
 - provide reuser with the concentrated wisdom of many preceding designers

285

Attribute-Based Architectural Styles (ABAS)

- Show how to reason about architectural decisions with respect to a specific quality attribute such as performance, security or reliability
- Associate a reasoning framework (qualitative or quantitative) with an architectural style
- Quality attribute based and attribute specific
- Design or analyze by considering one quality attribute at a time (one ABAS per quality, per style – an attribute model)
- Move architecture design one step closer to an engineering discipline

286

Quality Attributes

- Need a precise characterization of the quality attribute of concern
 - e.g., understand how to measure or observe modifiability and architectural decisions impact modifiability
- Quality attribute information: standard characterizations
 - external stimuli: events that cause the architecture to respond or change
 - architectural decisions: aspects (components, connectors, properties) that have a direct impact on achieving attribute responses
 - responses: architectural quality requirements expressed as measurable, observable quantities
- All ABASs for an attribute are organized around the same quality characterization

287

Example Use for Analysis and Design

- Use a handbook of ABASs to facilitate architecture design and analysis
- Sea Buoy problem
 - buoys collect temperature (10 seconds), wind speed (30 seconds) and location data (10 seconds) and broadcast (60 seconds) to air and sea traffic
 - on-demand broadcast 24-hour history (priority over periodic broadcasts)
 - SOS distress highest priority until reset
 - red light activate/deactivate
 - multiple sensors and accommodation for future sensor types
 - availability requirement?

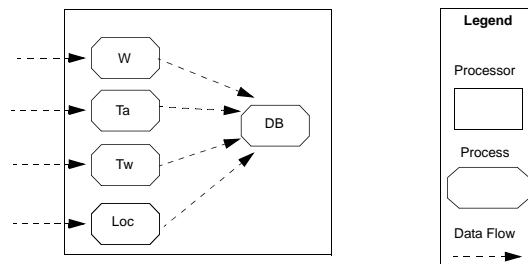
288

ABAS Used

- Qualities: performance, modifiability, availability
- Synchronization ABAS
 - mutually exclusive access to data repositories
- Concurrent Pipelines ABAS
 - several real-time information pipelines
- Abstract Data Repository ABAS
 - sensor and sensor type changes that result in data format changes or addition of new data types
- Simplex ABAS
 - analytical (modal) redundancy to achieve availability
- Incrementally incorporate different aspects of the problem into the architectural design
 - match function, topology and impact on attributes
 - to compose like-attribute ABASs, understand where decisions in one ABAS affect system properties also affected by decisions in another ABAS
 - to compose different-attribute ABASs, understand architectural decisions that affect both attributes

289

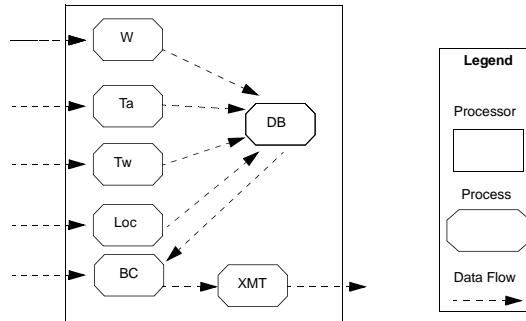
Synchronization ABAS



- Application: store sensor data then retrieve for periodic or on-demand transmission
- Stimuli: two or more periodic or sporadic input streams
- Response: end-to-end, worst-case latency
- Decisions: process relationships, prioritization, scheduling, data-locking policy

290

Synchronization with Broadcast and Transmit



- Broadcast has lower frequency, so assign it a lower priority
 - but Broadcast could block sensor access to DB

291

Apply Concurrent Pipeline ABAS

- Broadcast + Transmission pipeline
- Effective priority of a pipeline is strongly related to the lowest priority of all processes in the pipeline
 - If transmission process priority is lower than broadcast priority, it effectively lowers the priority of the entire pipeline

292

Add in History Function

- History request is stochastic (event-driven. aperiodic)
 - queuing analysis
- Synchronization and Concurrent pipeline analytical models were deterministic
 - rate monotonic analysis (RMA)
- Adjustments
 - adjust queuing model to account for preemption and synchronization
 - adjust RMA to account for random bursts of aperiodic arrivals

293

Add in SOS Function

- Constraint: SOS, history and broadcast all use the transmission process
 - SOS highest priority, then history, then broadcast
- Synchronization ABAS treated transmit as a critical section, thus a long history broadcast could block a high priority SOS
 - must make history broadcast preemptive
 - e.g. transmit long message in small chunks, preempt between chunks
 - (Note how Synchronization ABAS was extended for a special need)

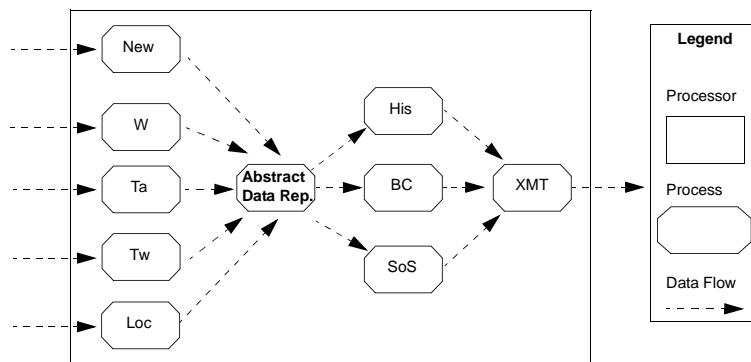
294

Adding in Modifiability

- Modifications: new sensor types
- Data Indirection ABAS with Abstract Data Repository sub-ABAS and Publish/Subscribe Sub-ABAS
- ABAS reasoning suggests these considerations
 - Will new sensors produce data in same or different format?
 - Are new sensors added or substitutes for existing sensors?
 - Is a new environmental parameter being sensed?
- Data Indirection results in pervasive change for new data format or for new environmental parameter
 - sub-ABASs accommodate the change
- No temporal ordering, so Abstract Data Repository sub-ABAS is acceptable

295

Current Architecture



- Consideration: data format conversion effect on process execution time, thus latency

296

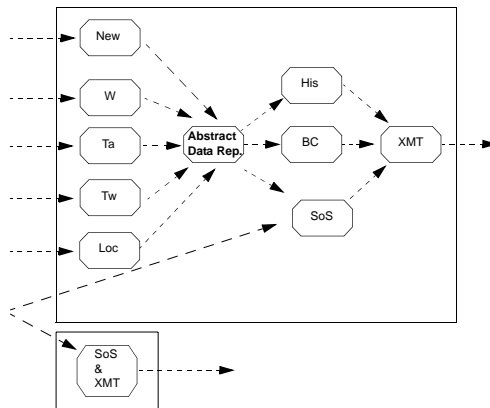
Trade-off Points for Different-Attribute ABAS

- Interaction across ABASs of different attribute types
 - understand how architectural decisions affect each quality attribute in isolation
 - look for situations architecture changes from the point of view of one attribute affect the parameters of functions for other attributes

297

Adding Availability

- SOS function is critical
- Simplex ABAS
 - add redundancy that use different mechanisms
- Provide a separate hardware unit for SOS transmit



298

The Five Viewpoints of RM-ODP

(Reference Model of Open Distributed Processing)
(ISO/IEC 10746-3:1996)

- Enterprise viewpoint
- Information viewpoint
- Computational viewpoint
- Engineering viewpoint
- Technology viewpoint

299

The Five Viewpoints of RM-ODP

- Enterprise Viewpoint
 - Concerns
 - The purpose, scope, and policies for an ODP system
 - Roles played by the system
 - Activities undertaken by the system
 - Policy statements about the system
 - Viewpoint Language (From the enterprise viewpoint, an ODP system and its environment are represented as a community of objects)
 - Enterprise objects composing the community
 - Roles fulfilled by each of those objects
 - Policies governing interactions between enterprise objects fulfilling roles
 - Policies governing the creation, usage, and deletion of resources by enterprise objects fulfilling roles
 - Policies governing the configuration of enterprise objects and assignment of roles to enterprise objects
 - Policies relating to environment contracts governing the system. With objects defined in terms of permissions, obligations, prohibitions, and behavior

300

The Five Viewpoints of RM-ODP

- Information Viewpoint
 - Concerns
 - The semantics of information and information processing in an ODP system
 - Viewpoint Language
 - The information language is defined in terms of three schemas:
 - Invariant schema: Predicates on objects that must always be true
 - Static schema: State of one or more objects at some point in time
 - Dynamic schema: Allowable state changes of one or more objects

301

The Five Viewpoints of RM-ODP

- Computational Viewpoint
 - Concerns
 - A functional decomposition of the system into objects that interact at interfaces
 - Viewpoint Language
 - The computational language covers concepts for:
 - Computational objects
 - Binding objects
 - Interactions
 - Interfaces

302

The Five Viewpoints of RM-ODP

- **Engineering Viewpoint**
 - **Concerns**
 - The mechanisms and functions required to support distributed interaction between objects in the system
 - **Viewpoint Language**
 - The engineering language includes concepts for:
 - Node structures
 - Channels that connect objects
 - Interfaces
 - Bindings
 - Relocation/migration
 - Clustering
 - Nodes
 - Failure types

303

The Five Viewpoints of RM-ODP

- **Technology Viewpoint**
 - **Concerns**
 - Captures the choice of technology in the system
 - How specifications are implemented
 - Specification of relevant technologies
 - Support for testing
 - **Viewpoint language**
 - The technology language addressed implementable standards and implementations.

304

Part VI

Achieving Optimum-Quality Results Selecting Kits and Frameworks Using Open Source vs. Commercial Infrastructures

*(See Presentation Sub-Topic 2:
Sample Enterprise Application Design)*

305

Part VII

Conclusion

306

Course Assignments

- Individual Assignments
 - Problems and reports based on case studies or exercises
- Project-Related Assignments
 - All assignments (other than the individual assessments) will correspond to milestones in the team project.
 - As the course progresses, students will be applying various methodologies to a project of their choice. The project and related software system should relate to a real-world scenario chosen by each team. The project will consist of inter-related deliverables which are due on a (bi-) weekly basis.
 - There will be only one submission per team per deliverable and all teams must demonstrate their projects to the course instructor.
 - A sample project description and additional details will be available under handouts on the course Web site.

307

Course Project

- Project Logistics
 - Teams will pick their own projects, within certain constraints: for instance, all projects should involve multiple distributed subsystems (e.g., web-based electronic services projects including client, application server, and database tiers). Students will need to come up to speed on whatever programming languages and/or software technologies they choose for their projects - which will not necessarily be covered in class.
 - Students will be required to form themselves into "pairs" of exactly two (2) members each; if there is an odd number of students in the class, then one (1) team of three (3) members will be permitted. There may not be any "pairs" of only one member! The instructor and TA(s) will then assist the pairs in forming "teams", ideally each consisting of two (2) "pairs", possibly three (3) pairs if necessary due to enrollment, but students are encouraged to form their own 2-pair teams in advance. If some students drop the course, any remaining pair or team members may be arbitrarily reassigned to other pairs/teams at the discretion of the instructor (but are strongly encouraged to reform pairs/teams on their own). Students will develop and test their project code together with the other member of their programming pair.

308

Sample Project Methodology

Very eXtreme Programming (VXP)

- After teams formed, 1/2 week to Project Concept
- 1/2 week to Revised Project Concept
- 2 to 3 iterations
- For each iteration:
 - 1/2 week to plan
 - 1 week to iteration report and demo

309

Sample Project Methodology

Very eXtreme Programming (VXP)

(continued)

- Requirements: Your project focuses on two application services
- Planning: User stories and work breakdown
- Doing: Pair programming, write test cases before coding, automate testing
- Demoing: 5 minute presentation plus 15 minute demo
- Reporting: What got done, what didn't, what tests show
- 1st iteration: Any
- 2nd iteration: Use some component model framework
- 3rd iteration: Refactoring, do it right this time

310

Revised Project Concept (Tips)

1. Cover page (max 1 page)
2. Basic concept (max 3 pages): Briefly describe the system your team proposes to build. Write this description in the form of either user stories or use cases (your choice). Illustrations do not count towards page limits.
3. Controversies (max 1 page)

311

First Iteration Plan (Tips)

- Requirements (max 2 pages):
- Select user stories or use cases to implement in your first iteration, to produce a demo by the last week of class
- Assign priorities and points to each unit - A point should correspond to the amount of work you expect one pair to be able to accomplish within one week
- You may optionally include additional medium priority points to do “if you have time”
- It is acceptable to include fewer, more or different use cases or user stories than actually appeared in your Revised Project Concept

312

First Iteration Plan (Tips)

- Work Breakdown (max 3 pages):
- Refine as *engineering tasks* and assign to pairs
- Describe specifically what will need to be coded in order to complete each task
- Also describe what unit and integration tests will be implemented and performed
- You may need additional engineering tasks that do not match one-to-one with your user stories/use cases
- Map out a *schedule* for the next weeks
- Be realistic – demo has to be shown before the end of the semester

313

2nd Iteration Plan (Tips): Requirements

- Max 3 pages
- Redesign/reengineer your system to use a component framework (e.g., COM+, EJB, CCM, .NET or Web Services)
- Select the user stories to include in the new system
 - Could be identical to those completed for your 1st Iteration
 - Could be brand new (but explain how they fit)
- Aim to maintain project velocity from 1st iteration
- Consider what will require new coding vs. major rework vs. minor rework vs. can be reused “as is”

314

2nd Iteration Plan (Tips): Breakdown

- Max 4 pages
- Define engineering tasks, again try to maintain project velocity
- Describe new unit and integration testing
- Describe regression testing
 - Can you reuse tests from 1st iteration?
 - If not, how will you know you didn't break something that previously worked?
- 2nd iteration report and demo to be presented before the end of the semester

315

2nd Iteration Report (Tips): Requirements

- Max 2 pages
- For each engineering task from your 2nd Iteration Plan, indicate whether it succeeded, partially succeeded (and to what extent), failed (and how so?), or was not attempted
- Estimate how many user story points were actually completed (these might be fractional)
- Discuss specifically your success, or lack thereof, in porting to or reengineering for your chosen component model framework(s)

316

2nd Iteration Report (Tips): Testing

- Max 3 pages
- Describe the general strategy you followed for unit testing, integration testing and regression testing
- Were you able to reuse unit and/or integration tests, with little or no change, from your 1st Iteration as regression tests?
- What was most difficult to test?
- Did using a component model framework help or hinder your testing?

317

Project Presentation and Demo

- All Iterations Due
- Presentation slides (optional)

318

Readings

- Readings
 - Slides and Handouts posted on the course web site
 - Documentation provided with business and application modeling tools (e.g., Popkin Software Architect)
 - SE Textbook: Chapters 8-12 (Part 2), 18-19 (Part 3), 30 (Part 5)
- Project Frameworks Setup (ongoing)
 - As per references provided on the course Web site
- Individual Assignment
 - See Session 6 Handout: “Assignment #4”
- Team Assignment
 - See Session 6 Handout: “Team Project” (Part 2)

319

Next Session:

From Analysis and Design to Software Architectures (Part II)

- OOAD Using UML
- Micro/Macro Architecture
- Design Patterns
- Architectural Patterns
- Enterprise Architectural Patterns
- Sample Middleware Reference Architectures
- Architectural Capabilities
- Object-Oriented Design Guidelines
- Summary
 - Individual Assignment #5
 - Project (Part 2)

320