14 - Architecture, Frameworks, Middleware

- Reading & Sources
  - David Garlan, “Software Architecture: a Roadmap,” Proceedings of the conference on The future of Software engineering, Limerick, Ireland, June 04 - 11, 2000
  - M. Shaw and P. Clements,”A field guide to boxology: Preliminary classification of architectural styles for software systems,” Proceedings of COMPSAC 1997, August 1997
  - M. Shaw and D. Garlan, Tutorial Slides on Software Architecture http://www-2.cs.cmu.edu/afs/cs/project/tinker-arch/www/html/Tutorial_Slides/Soft_Arch/quick_index.html
  - Garlan, David & Shaw, “An Introduction To Software Architecture,” Technical report, The Software Engineering Institute, Carnegie Mellon University
  - Ralph E. Johnson, “Frameworks= (Components + Patterns).”Communications of the ACM, October 1997 Vol. 40, No. 10

Software Architectures

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

Requirements

High-level Design

Detailed Design

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2004

taxonomy

dataflow
- problem can be decomposed into sequential stages
- involves transformations on continuous streams of data

call/return
- problem difficult to model
- anticipation of many changes
- reuse

independent components
- flexibility-configurability, loose coupling, reactive tasks
- Styles: Heatbeat, Prod-Con, Client-Server, token-passing

implicit invocation | **explicit invocation**

virtual machine
- simulate functionality which is not native
- execution engine SW “implemented”

data-centered
- central issue is understanding the data
- DB: highly structured & dynamic
- BB: noisy environment

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2004

Architectural taxonomy (“boxology”)

- dataflow
  - batch sequential
  - data flow network
  - pipes & filter
- call/return
  - main program/subroutines
  - abstract data types
  - objects
  - call based client/server
  - layered
- independent components
  - communicating processes
  - distributed
  - event systems (implicit, explicit)
- virtual machine
  - interpreter
  - rule-based
- data-centered
  - repository
  - blackboard

can decompose into sequential stages
involves transformations on continuous (or on very long streams) streams of data

flexibility, configurability, loose coupling
hierarchies, producer-consumer, tightly connected

cross-platform
late decision on hardware

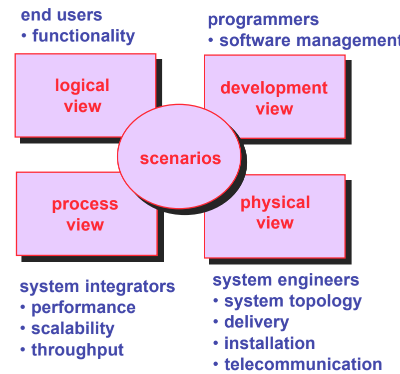
focus on management and representation of data
long-lived (persistent) data is focus on repositories
stream of incoming requests to access highly structured data
changing data
“noisy” input data, uncertain execution order can not be predetermined, consider a blackboard

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2004

Views

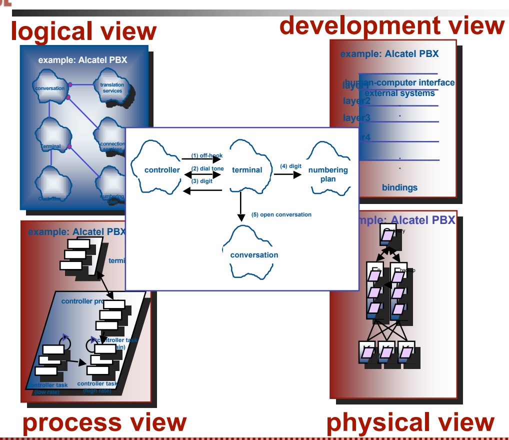
- What is a view?
  - A view is a presentation of a model, which is a complete description of a system from a particular perspective.
- Proposed views:
  - Logical View - captures the object model
  - Process View - captures the concurrency and synchronization aspects
  - Development View - captures static organization of the software in its development environment
  - Physical View - captures the way software is mapped on hardware
  - The “4+1” view: these plus scenarios

4+1 view of software architecture



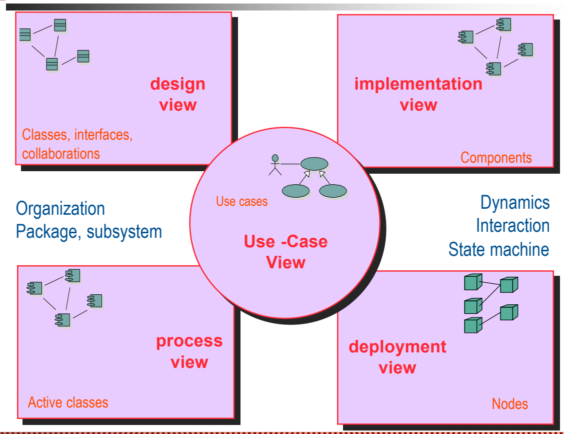
University of Massachusetts Amherst · Department of Computer Science · CMPSCI 520/620 Fall 2004

4+1 views



University of Massachusetts Amherst · Department of Computer Science · CMPSCI 520/620 Fall 2004

The Rational 4+1 Views



University of Massachusetts Amherst · Department of Computer Science · CMPSCI 520/620 Fall 2004

The Rational 4+1 Views

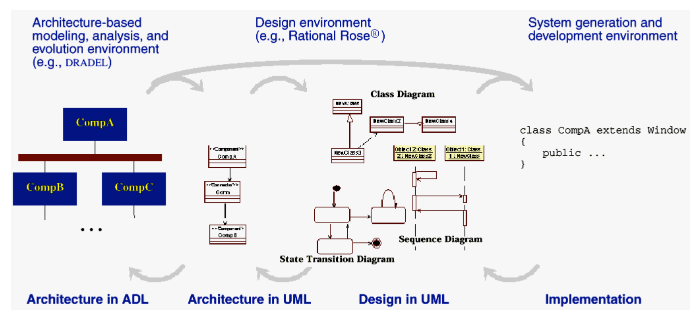
UML SW Development Life Cycle

- Use-case driven
  - use cases are used as a primary artifact for establishing the desired behavior of the system, for verifying and validating the system's architecture, for testing, and for communicating among the stakeholders of the project
- Architecture-centric
  - a system's architecture is used as a primary artifact for conceptualizing, constructing, managing, and evolving the system under development
- Iterative
  - one that involves managing a stream of executable releases
- Incremental
  - one that involves the continuous integration of the system's architecture to produce these releases

Architectural View Mismatches in UML

- Different UML diagrams present different system views
  - redundant information across views
- Key challenge is to ensure inter-view consistency
- Ramifications on round-trip engineering

Round-Trip Software Engineering Using UML



Nenad Medvidovic Assessing the Suitability of UML for Modeling Software Architectures

Architecture Description Languages

- formal notations for representing and analyzing architectural designs
- provide both a conceptual framework and a concrete syntax for characterizing software architectures
- tools for parsing, displaying, compiling, analyzing, or simulating architectural descriptions.

ADL Examples

- Adage
  - supports the description of architectural frameworks for avionics navigation and guidance
- Aesop
  - supports the use of architectural styles
- C2
  - supports the description of user interface systems using an event-based style
- Darwin
  - supports the analysis of distributed message-passing systems
- Meta-H
  - provides guidance for designers of real-time avionics control software;
- Rapide
  - allows architectural designs to be simulated, and has tools for analyzing the results of those simulations;
- SADL
  - provides a formal basis for architectural refinement;
- UniCon
  - has a high-level compiler for architectural designs that supports a mixture of heterogeneous component and connector types;
- Wright
  - supports the formal specification and analysis of interactions between architectural components.

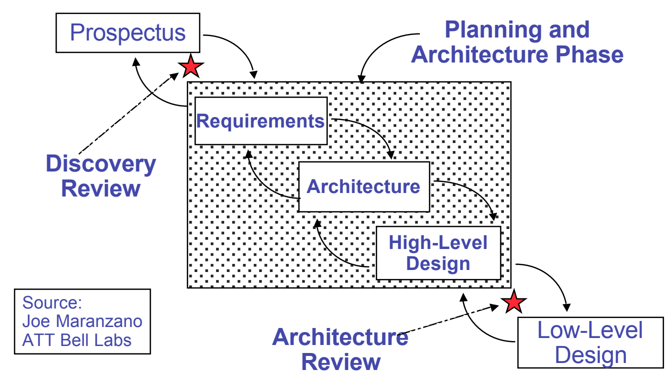
formal architectural specification.

- module interconnection languages
  - static aspects of component interaction
  - definition and use of types, variables, and functions among components
  - examples: INTERCOL, PIC, CORBA/IDL
- process algebras
  - dynamic interplay among components
  - concerned with the protocols by which components communicate
  - examples: Wright (based on CSP), Chemical Abstract Machine (based on term rewriting)
- event languages
  - identification and ordering of events
  - event is a very flexible, abstract notion
  - example: Rapide

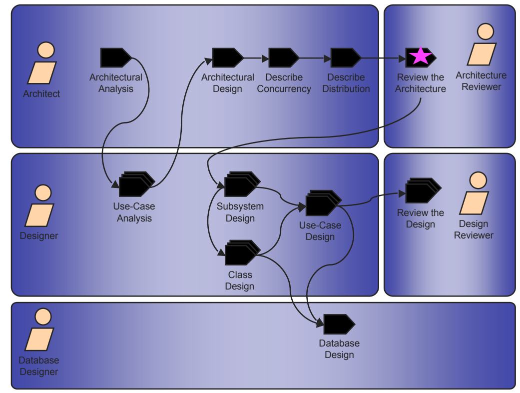
Evaluation & analysis

- conduct a formal review with external reviewers
  - time the evaluation to best advantage
  - choose an appropriate evaluation technique
  - create an evaluation contract
  - limit the number of qualities to be evaluated
  - insist on a system architect
- benefits
  - financial
  - increased understanding and documentation of the system
  - detection of problems with the existing architecture
  - clarification and prioritization of requirements
  - organizational learning

Architectural Design Reviews



Rational Unified Process



Benefits

- examples
  - AT&T
    - 10% reduction in project costs, on projects of 700 staff days or longer, the evaluation pays for itself.
  - consultants
    - reported 80% repeat business, customers recognized sufficient value
  - where architecture reviews did not occur
    - customer accounting system estimated to take two years, took seven years, re-implemented three times, performance goals never met
    - large engineering relational database system, performance made integration testing impossible, project was cancelled after twenty million dollars had been spent.

Architecture vs Frameworks

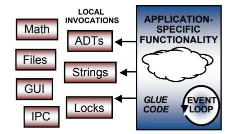
- Frameworks
  - an object-oriented reuse technique
  - used successfully for some time & are an important part of the culture of long-time object-oriented developers,
  - BUT they are not well understood outside the object-oriented community and are often misused
- Question:
  - are frameworks mini-architectures,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

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

Frameworks & Class Libraries

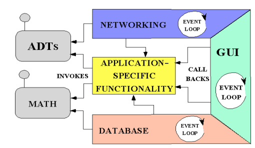
- developers often do not even know they are using a framework, but refer to a “class library”
- frameworks differ from other class libraries by reusing high-level design
  - more to learn before a class can be reused
  - can never be reused in isolation; typically a set of classes must be learned at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

Frameworks & Class Libraries



Class Library Architecture

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



Framework Architecture

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

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

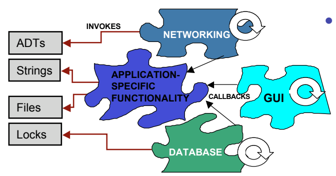
Components & frameworks

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

Components & frameworks

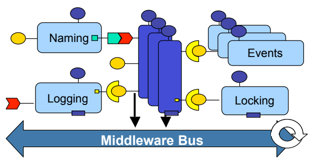
- frameworks
  - provide a reusable context for components
  - provide a standard way for components to handle errors, to exchange data, and to invoke operations on each other
    - “component systems” such as OLE, OpenDoc, and Beans, are really frameworks that solve standard problems that arise in building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

Frameworks & Components



Framework Architecture

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



Component Architecture

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

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

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2004

Comparison

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

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2004

Frameworks as Reusable Design

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

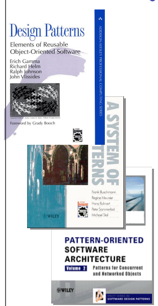
UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2004

Frameworks and domain-specific architectures

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

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2004

Overview of Patterns



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

software patterns

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

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2004

qualities of a pattern

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

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2004

Taxonomy of Patterns & Idioms

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

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2004

Frameworks and Patterns

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

Frameworks and Patterns

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

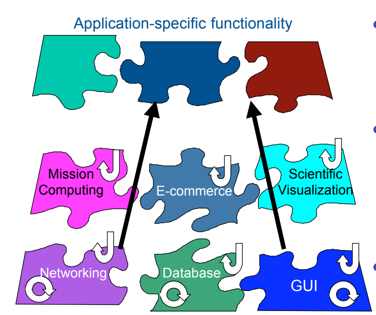
University of Massachusetts Amherst • Department of Computer Science • CMPSCI 520/620 Fall 2004

Frameworks

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

University of Massachusetts Amherst • Department of Computer Science • CMPSCI 520/620 Fall 2004

Framework Characteristics



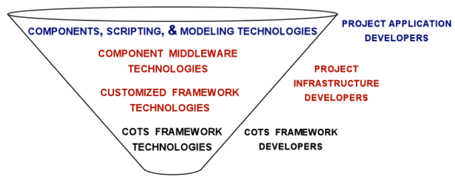
- Frameworks exhibit "inversion of control" at runtime via callbacks
- Frameworks provide integrated domain-specific structures & functionality
- Frameworks are "semi-complete" applications

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

University of Massachusetts Amherst • Department of Computer Science • CMPSCI 520/620 Fall 2004

Using Frameworks Effectively

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

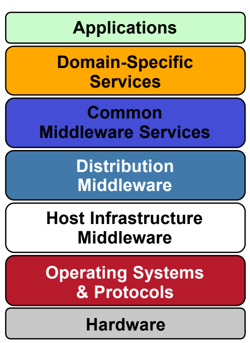


Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

Relation to Middleware

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

Evolution of Middleware



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

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

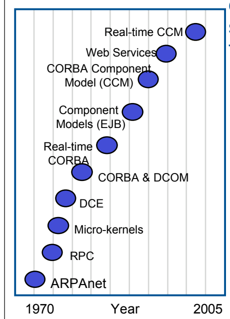
Middleware

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

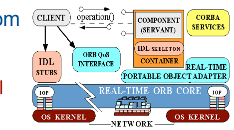
Middleware

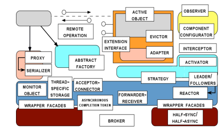
- Common middleware services
  - augments the distribution middleware by defining domain-independent services, such as event notifications, logging, multimedia streaming, persistence, security, transactions, fault tolerance, and distributed concurrency control
  - applications can reuse these services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

Progress



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





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

Design

- Readings
  - David Parnas "On the Criteria To Be Used in Decomposing Systems into Modules," Comm. ACM 15, 12 (Dec. 1972), 1053-1058
  - David Parnas"On a 'Buzzword': Hierarchical Structure" IFIP Congress '74. North Holland Publishing Company, 1974 pp. 336-339
  - David Parnas"On the design and development of program families" IEEE Trans. On SE., vol. SE-2, pp.1-9, Mar. 1976

History of Software Design

- 1960s
  - Structured Programming
    - ("Goto Considered Harmful", E.W.Dijkstra)
    - Emerged from considerations of formally specifying the semantics of programming languages, and proving programs satisfy a predicate.
    - Adopted into programming languages because it's a better way to think about programming
- 1970s
  - Structured Design
    - Methodology/guidelines for dividing programs into subroutines.
- 1980s
  - Modular (object-based) programming
    - Ada, Modula, Euclid, …
    - Grouping of sub-routines into modules with data.
- 1990s
  - Object-Oriented Languages started being commonly used
  - Object-Oriented Analysis and Design for guidance.

Key Word In Context

"The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters.

Any line may be 'circularly shifted' by repeatedly removing the first word and appending it at the end of the line.

The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order."

On the Criteria for Decomposing Systems into Modules. David Parnas. *CACM*, 1972

KWIC example "borrowed" from Software Architectures © David Garlan

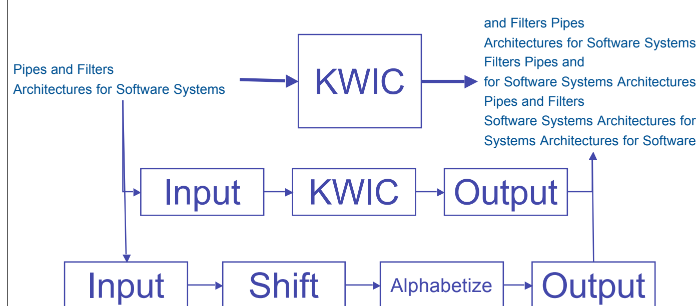
KWIC: Key Word In Context

- Inputs: Sequence of lines
  - Pipes and Filters
  - Architectures for Software Systems
- Outputs: Sequence of lines, circularly shifted and alphabetized
  - and Filters Pipes
  - Architectures for Software Systems
  - Filters Pipes and
  - for Software Systems Architectures
  - Pipes and Filters
  - Software Systems Architectures for
  - Systems Architectures for Software

Design Considerations

- Change in Algorithm
  - e.g., batch vs. incremental
- Change in Data Representation
  - e.g., line storage, explicit vs implicit shifts
- Change in Function
  - e.g., eliminate lines starting with trivial words
- Performance
  - e.g., space and time
- Reuse
  - e.g., sorting

Stepwise Refinement Strategy



Solution 1

- Decompose the overall processing into a sequence of processing steps.
  - Read lines; Make shifts; Alphabetize; Print results
- Each step transforms the data completely.
- Intermediate data stored in shared memory.
  - Arrays of characters with indexes
  - Relies on sequential processing

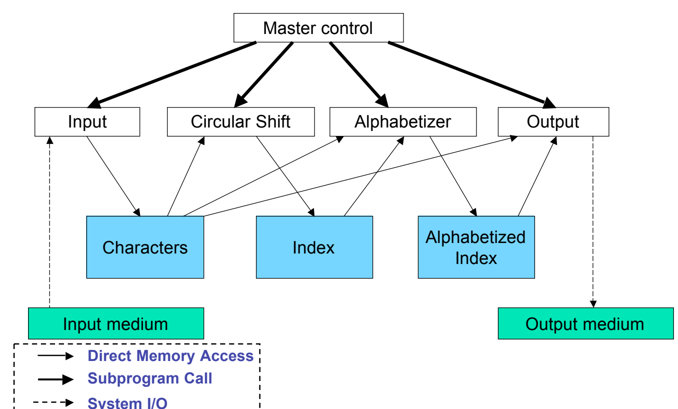
Solution 1: Modularization

- Module 1: Input
  - Reads data lines and stores them in “core”.
  - Storage format: 4 chars/machine word; array of pointers to start of each line.
- Module 2: Circular Shift
  - Called after Input is done.
  - Reads line storage to produce new array of pairs:
    - (index of 1st char of each circular shift,
    - index of original line)
- Module 3: Alphabetize
  - Called after Circular Shift.
  - Reads the two arrays and produces new index.

Solution 1

- Module 4: Output
  - Called after alphabetization and prints nicely formatted output of shifts
  - Reads arrays produced by Modules 1 & 3
- Module 5: Master Control
  - Handles sequencing of other modules
  - Handles errors

KWIC Modularization 1



Properties of Solution 1

- Batch sequential processing.
- Uses shared data to get good performance.
- Processing phases handled by control module.
  - So has some characteristics of main program - subroutine organization.
  - Depends critically on single thread of control.
- Shared data structures exposed as inter-module knowledge.
  - Design of these structures must be worked out before work can begin on those modules.

Advantages & Disadvantages

- Advantages
  - computations can share the same storage
  - allow efficient data representation
  - has a certain intuitive appeal
    - distinct computational aspects are isolated in different modules
- Disadvantages
  - serious drawbacks in terms of its ability to handle changes
    - a change in data storage format will affect almost all of the modules
    - changes in algorithm and enhancements to system function are not easily handled
  - reuse is not well-supported because each module of the system is tied tightly to this particular application.

Criteria for decomposition

- Modularization 1
  - Each major step in the processing was a module
- Modularization 2
  - Information hiding
    - Each module has one or more "secrets"
    - Each module is characterized by its knowledge of design decisions which it hides from all others.
  - Lines
    - how characters/lines are stored
  - Circular Shifter
    - algorithm for shifting, storage for shifts
  - Alphabetizer
    - algorithm for alpha, laziness of alpha

Solution 2

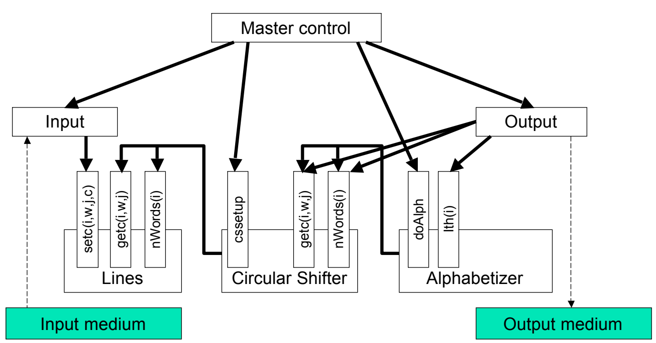
- Maintain same flow of control, but
- Organize solution around set of data managers (objects):
  - for initial lines
  - shifted lines
  - alphabetized lines
- Each manager:
  - handles the representation of the data
  - provides procedural interface for accessing the data

Solution 2: Modularization

- Module 1: Line storage
  - Manages lines and characters; procedural interface
  - Storage format: not specified at this point
- Module 2: Input
  - Reads data lines and stores using "Line Storage"
- Module 3: Circular Shift
  - Provides access functions to characters in circular shifts
  - Requires CSSETUP as initialization after Input is done

- Module 4: Alphabetize
  - Provides index of circular shift
  - ALPH called to initialize after Circular Shift
- Module 5: Output
  - Prints formatted output of shifted lines
- Module 6: Master Control
  - Handles sequencing of other modules

KWIC Modularization 2



Properties of Solution 2

- Module interfaces are abstract
  - hide data representations
    - could be array + indices, as before
    - or lines could be stored explicitly
  - hide internal algorithm used to process that data
    - could be lazy or eager evaluation
  - require users to follow a protocol for correct use
    - initialization
    - error handling
- Allows work to begin on modules before data representations are designed.
- Could result in same executable code as first solution.
  - according to Parnas, at least

Comparisons

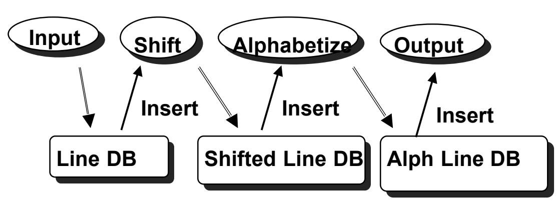
- Change in Algorithm
  - Solution 1: batch algorithm wired into
  - Solution 2: permits several alternatives
- Change in Data Representation
  - Solution 1: Data formats understood by many modules
  - Solution 2: Data representation hidden
- Change in Function
  - Solution 1: Easy if add a new phase of processing
  - Solution 2: Modularization doesn't give particular help

Independent Development

- Modularization 1
  - Must design all data structures before parallel work can proceed
  - Complex descriptions needed
- Modularization 2
  - Must design interfaces before parallel work can begin
  - Simple descriptions only
- Comprehensibility
  - Modularization 2 is better
    - Parnas subjective judgment

KWIC: Solution 3 (Toolies)

Interactive Version



Advantage:
Tool separation makes function enhancements easier.

→ Proc Call
→ Events

Summary

- Every architect should have a standard set of architectural styles in his/her repertoire
  - it is important to understand the essential aspects of each style: when and when not to use them
  - examples: pipe and filters, objects, event-based systems, blackboards, interpreters, layered systems
- Choice of style can make a big difference in the properties of a system
  - analysis of the differences can lead to principled choices among alternatives