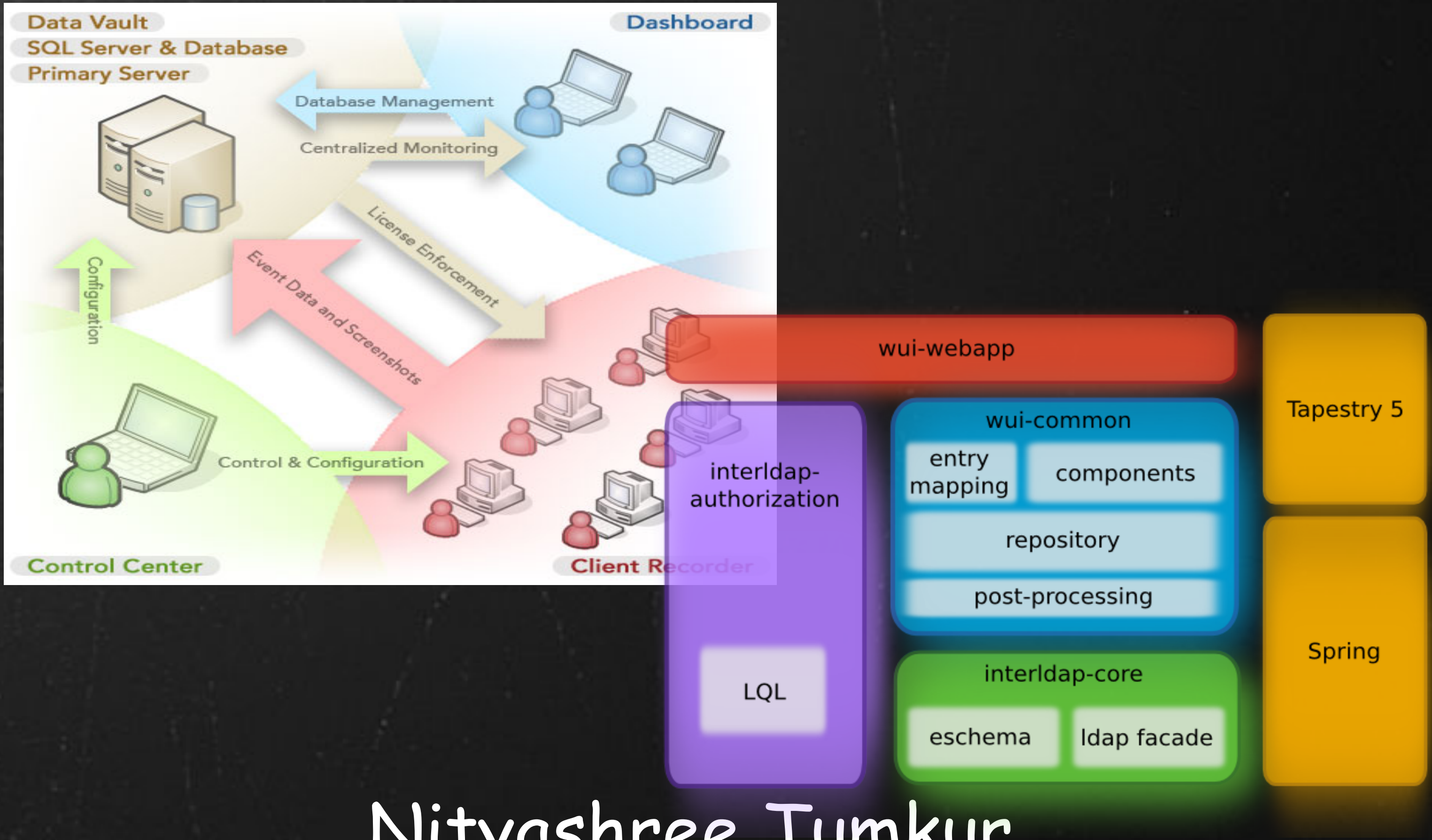
ISET62- SOFTWARE ARCHITECTURE

M.Sc.(SE) – III Year – VI Semester

Unit I to Unit V - Notes

Software Architecture



wui-webapp

Tapestry 5

interldap-authorization

LQL

wui-common

entry mapping

components

repository

post-processing

interldap-core

eschema

ldap facade

Spring

Nityashree Tumkur
Samyukta Mudugal

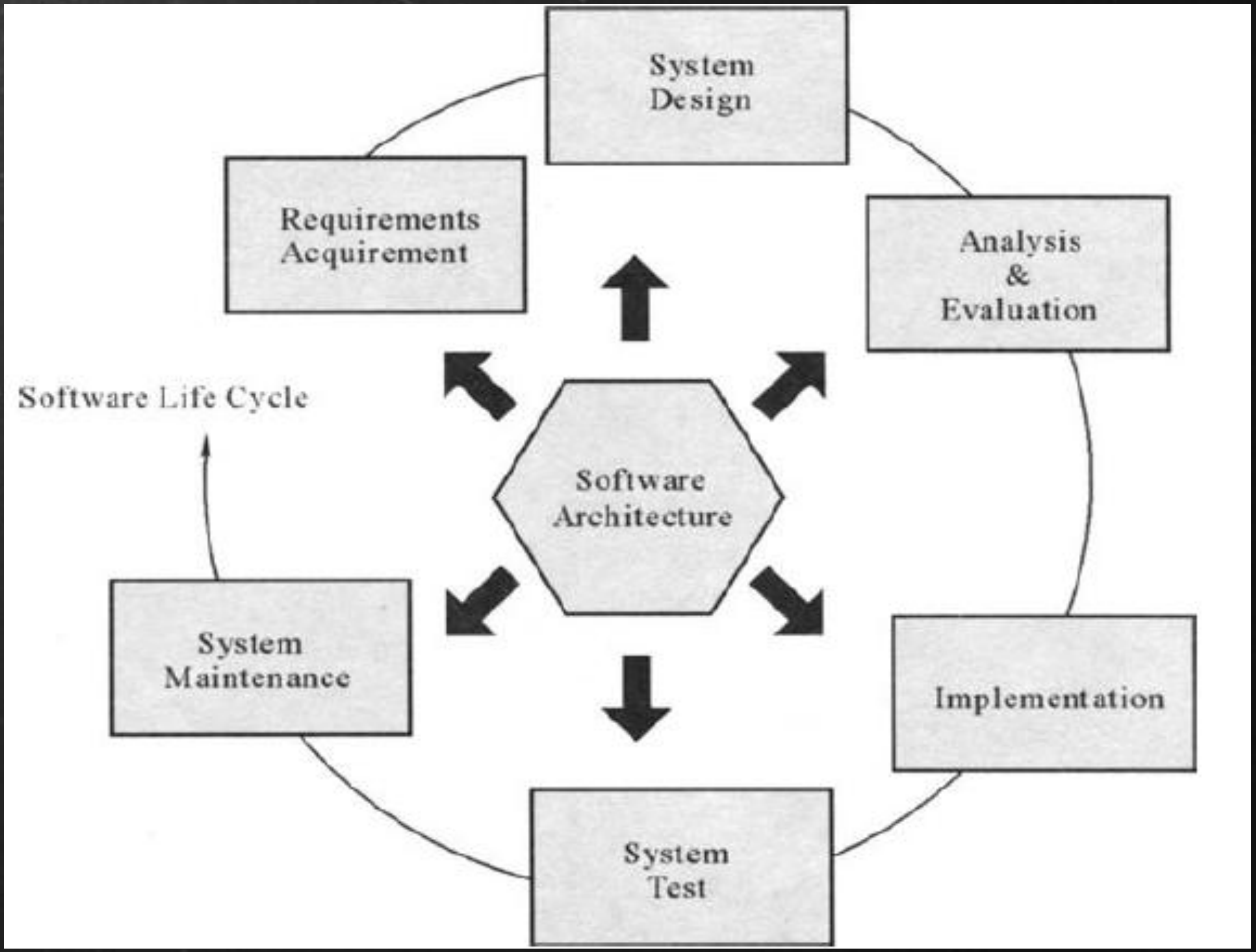
What is Software Architecture?

It is the structure of the system which consists of software components, the externally visible properties of those components and the relationship between them.

Features

- Every System has its own architecture but they are not identical.
- Software architecture and its description are different.
- The different stakeholders are

  1. Users of the System
  2. Acquirers of the System
  3. Developers of the System
  4. Maintainers of the System

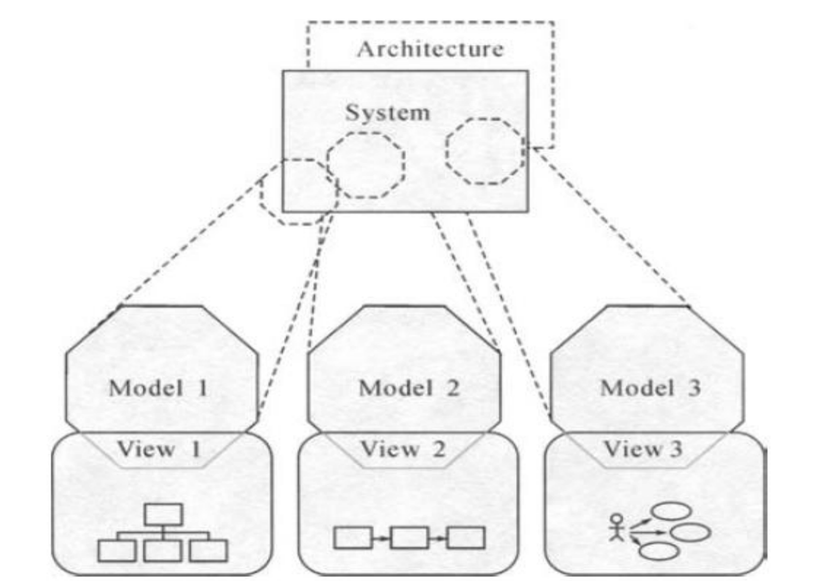


Architecture Centered Life Cycle

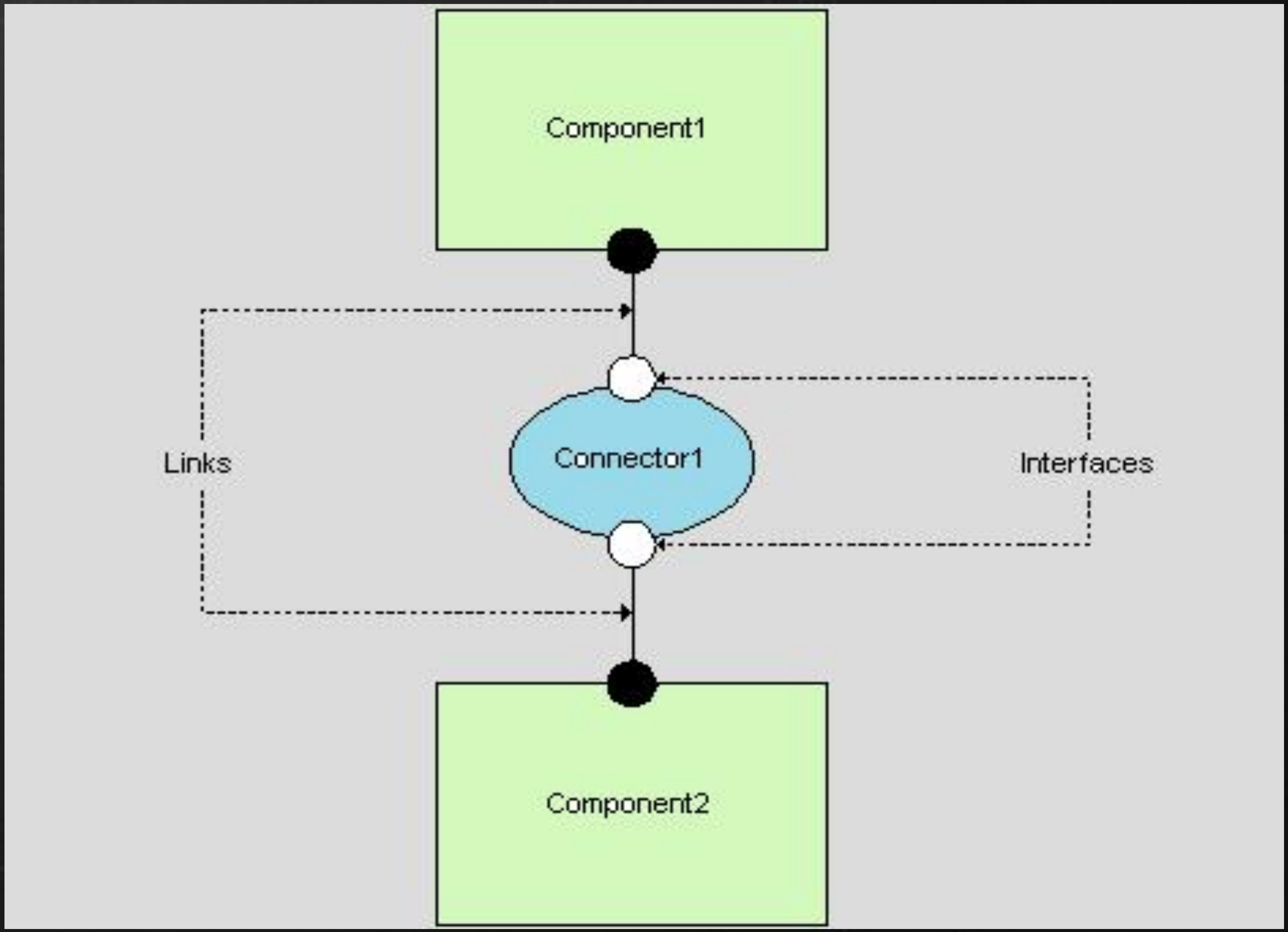
Views Used in Software Architecture

- Software architecture is organised in views which are analogous different types of blueprints made in building architecture.

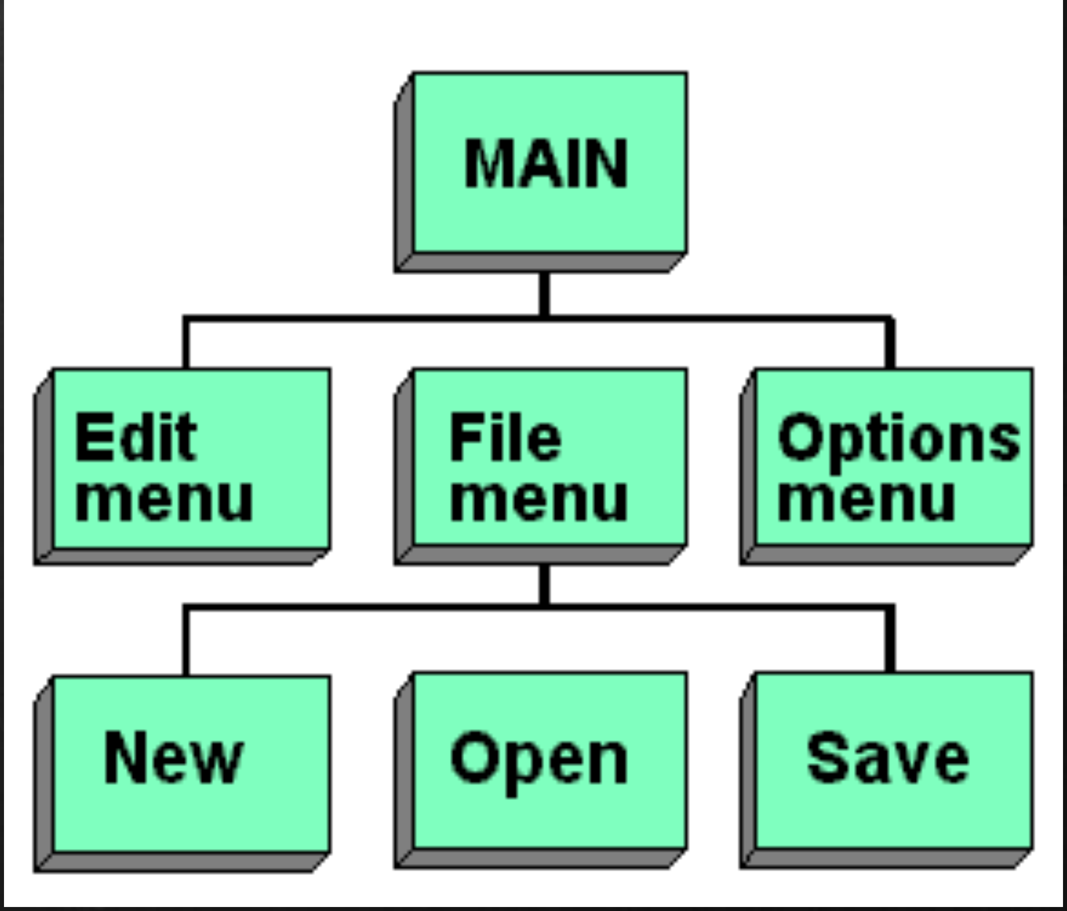
- Different views are:
  - Component and Connect
  - Decomposition view.
  - Allocation view



Component - Connector View



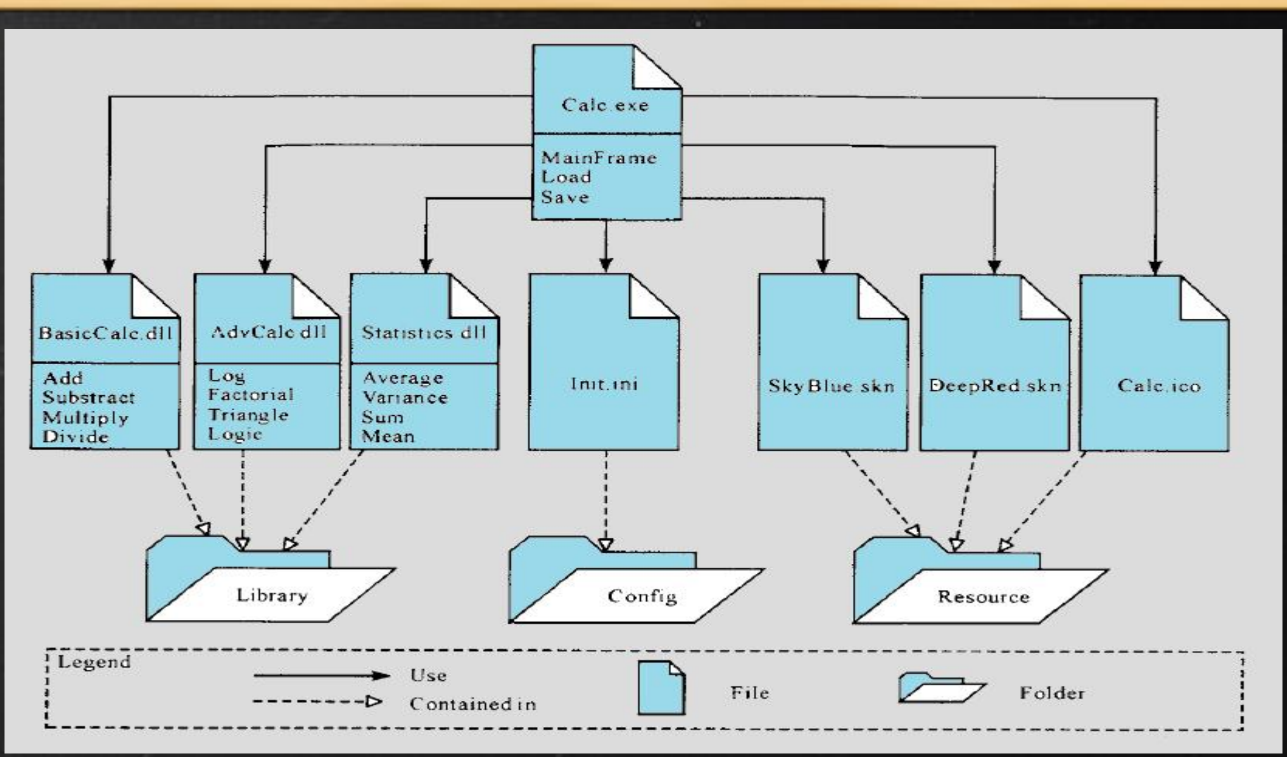
Decomposition View

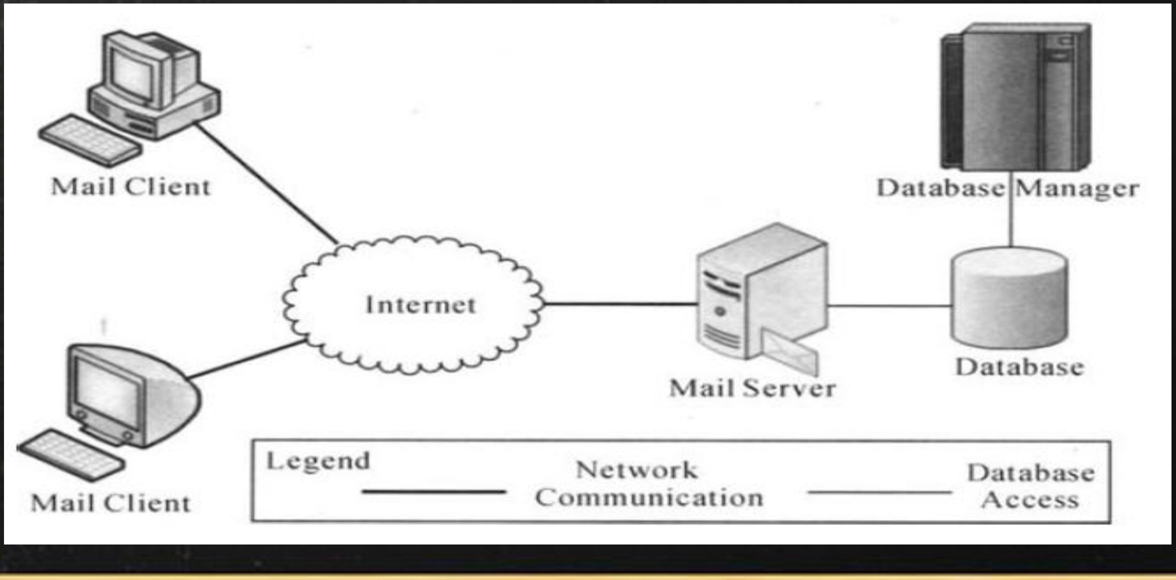


From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.

Allocation View

Implementation view





Deployment View

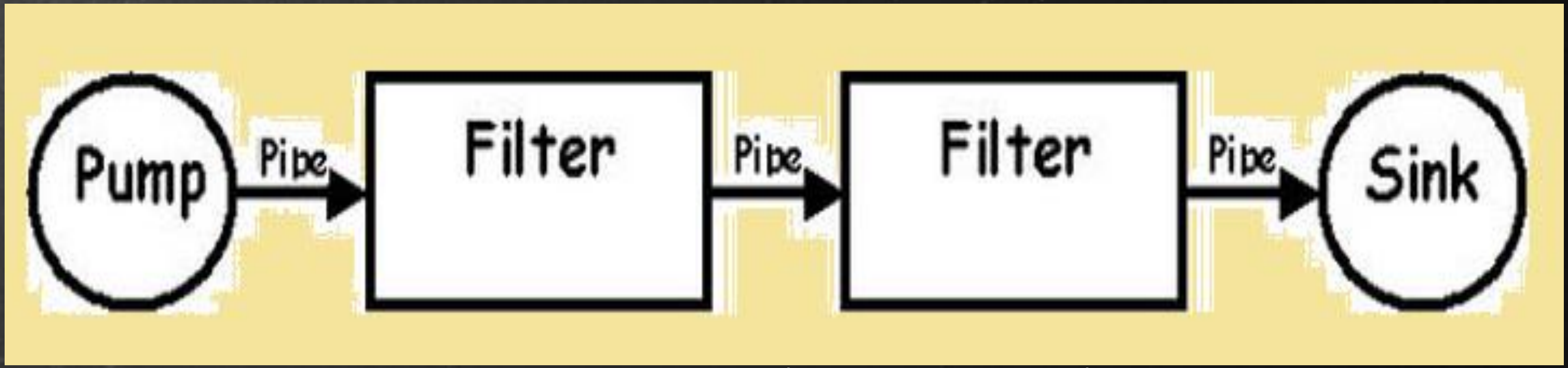
Architectural Styles

- Pipes & Filters
- Client- Server
- Event Driven
- Hierarchical Layer
- Data Sharing
- Object Oriented

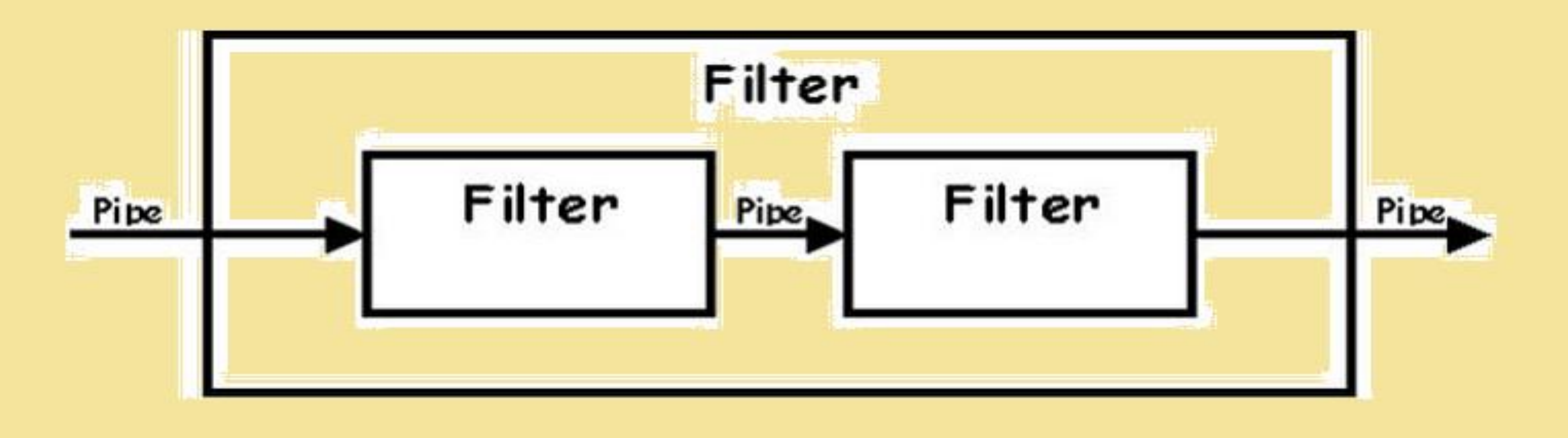
Pipes & Filters

- Very Simple yet powerful and robust architecture.
- Examples:
  1. Unix Programs
  2. Compilers
- Components
  1. Pipe
  2. Filter
  3. Pump
  4. Sink

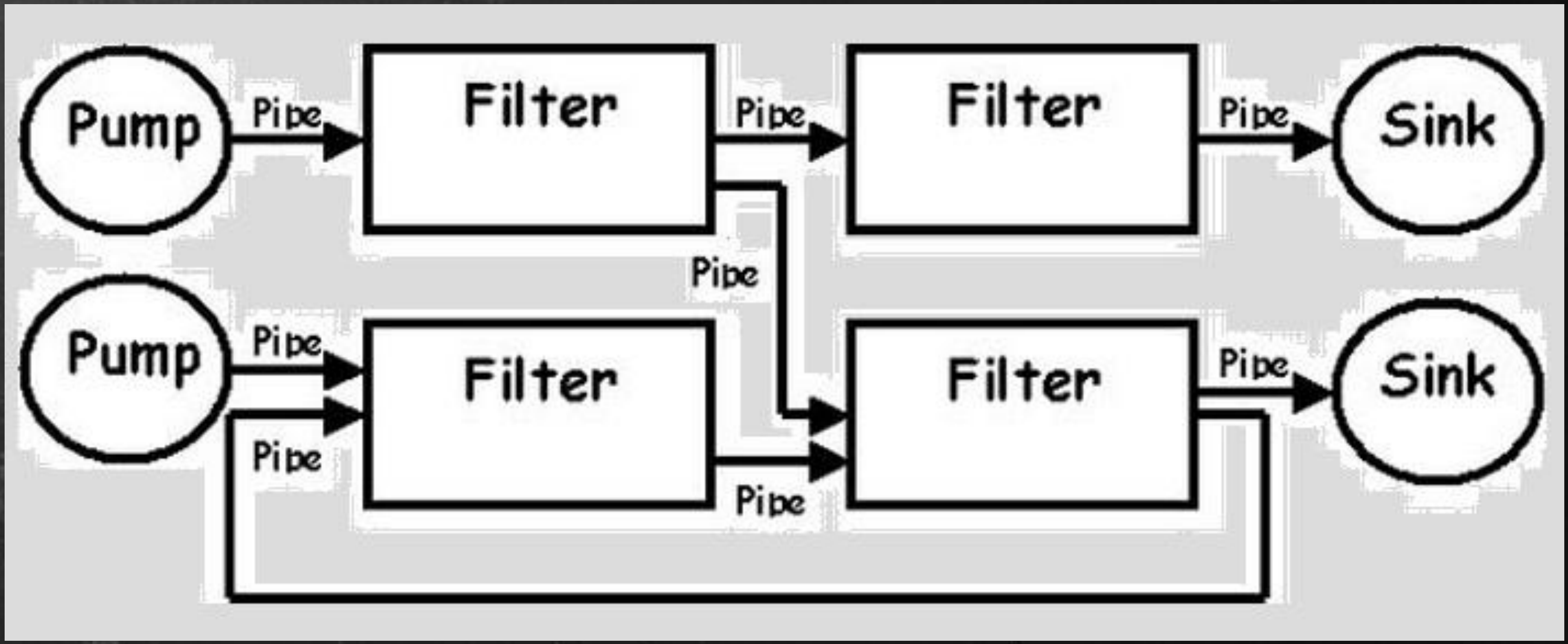
FILTER
...finally I found a good use for that leak!
search ID: pwen220
© Original Artist
Reproduction rights obtainable from
www.CartoonStock.com



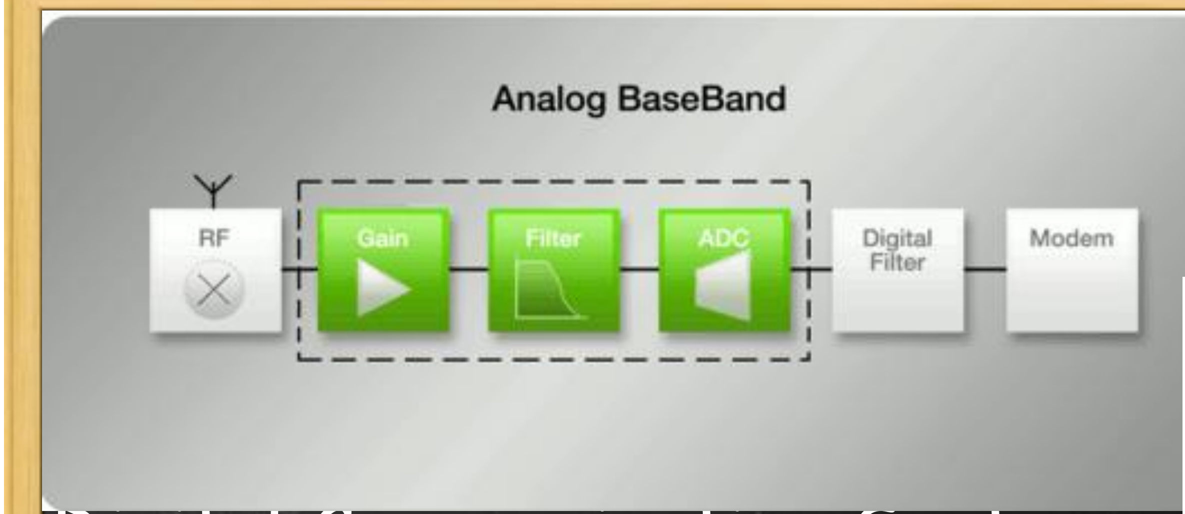
Pipes & Filters Style



Recursive Technique

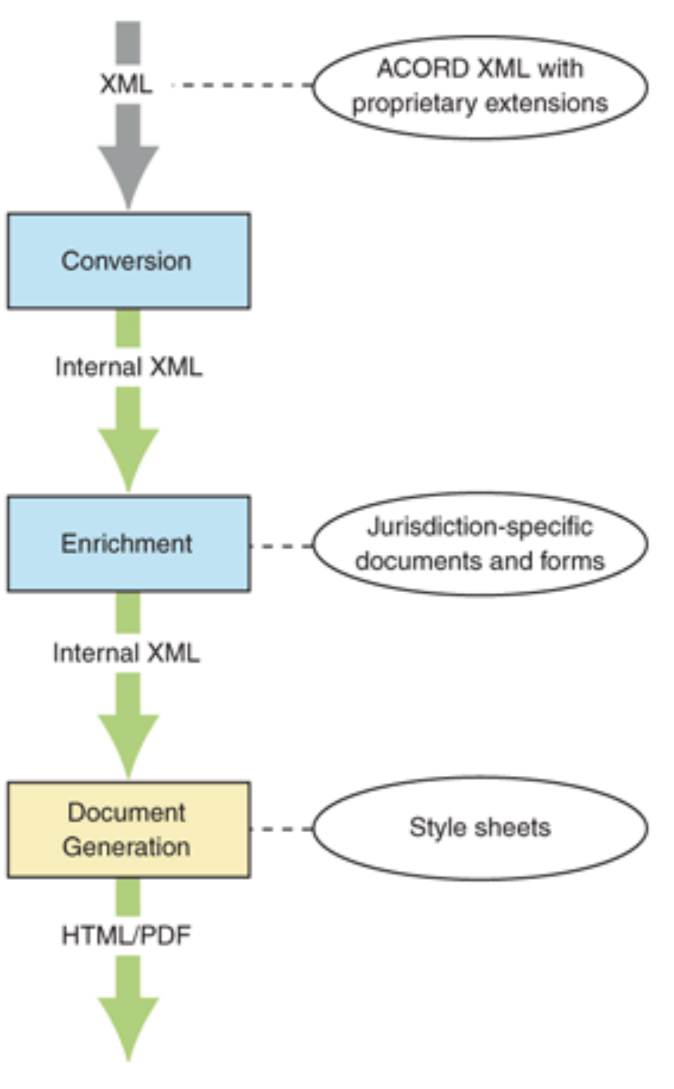


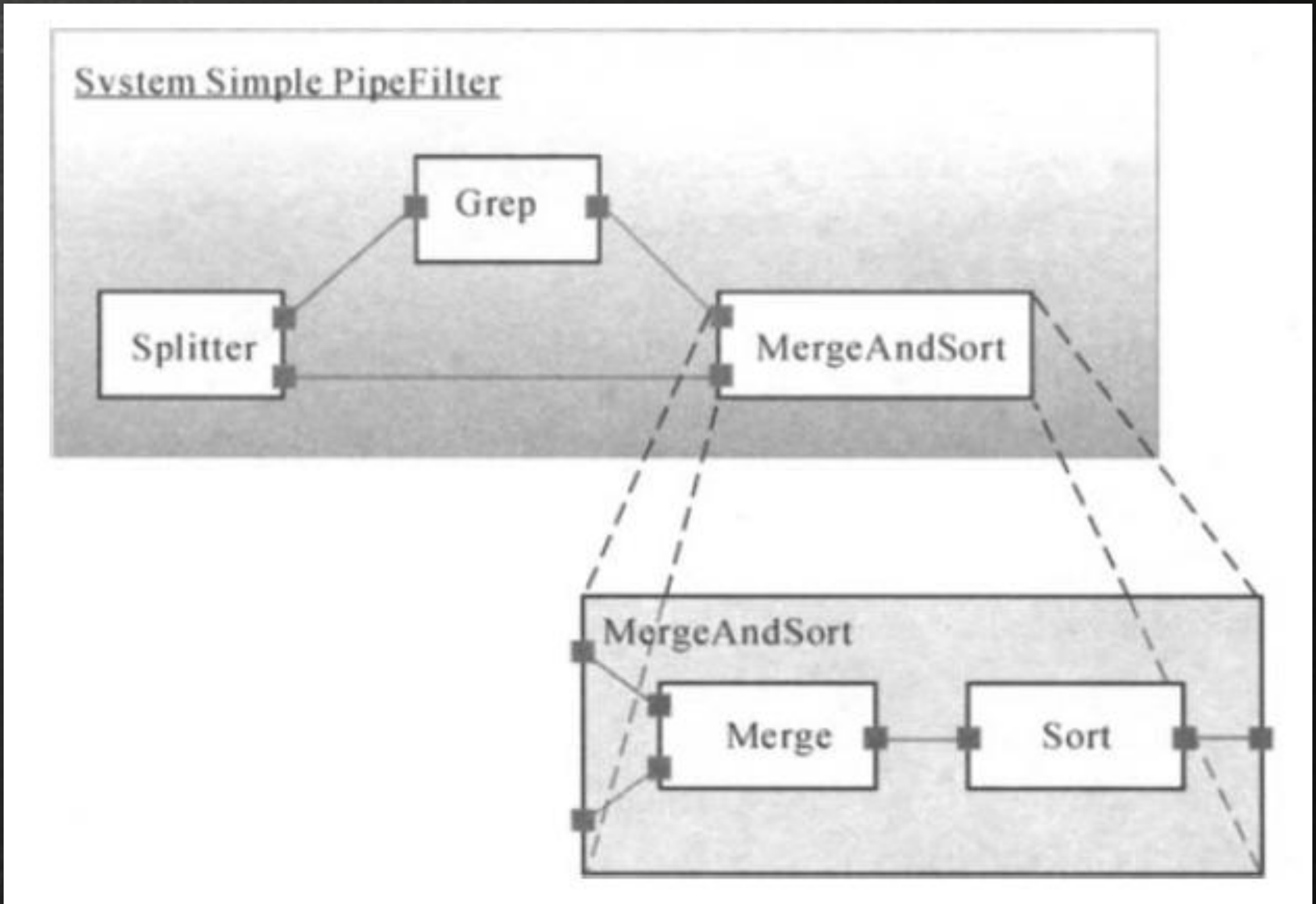
Relationship between different filter processes.



Digital Communication System

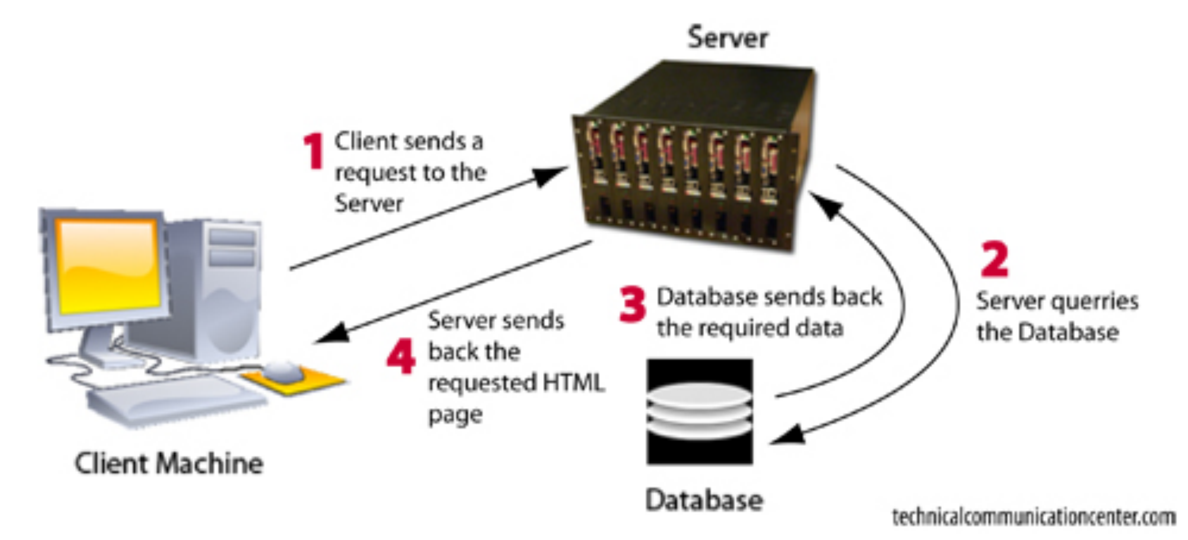
Web Application





Another Example

Client Server Style


Server
1 Client sends a request to the Server
2 Server querries the Database
3 Database sends back the required data
4 Server sends back the requested HTML page
Client Machine
Database
technicalcommunicationcenter.com

Distributed Application Architecture that partitions the tasks into service providers and service requesters

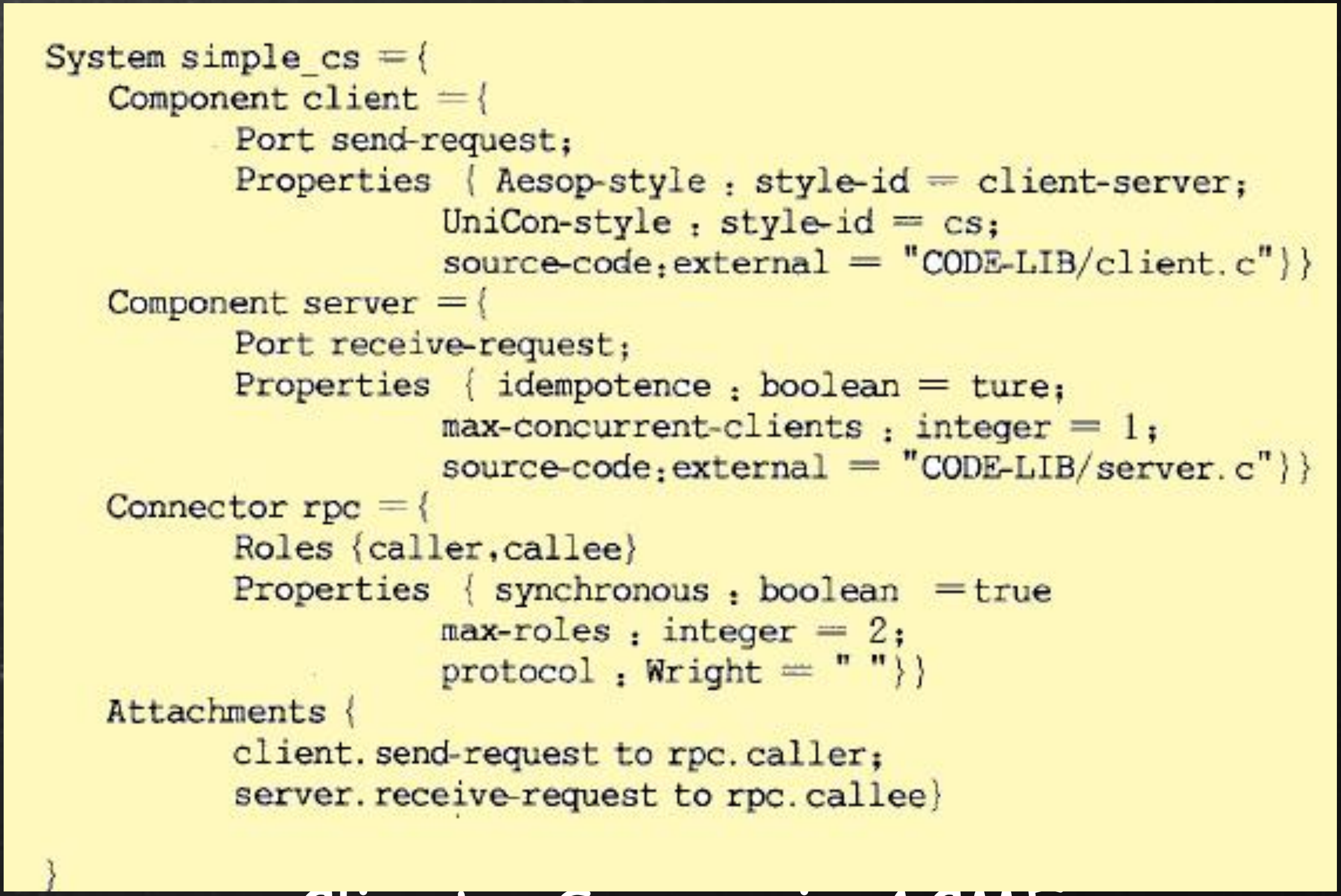
Advantages

- Roles and responsibilities of computing systems to be distributed among independent computers known to each other only through the network.
- All the data is stored in the server which have better security controls.
- Caters to multiple different clients with different capabilities.
- Data updates are easier and faster as Data is centralized.

Disadvantages

- As the number of client requests increases the server becomes overloaded
- Client - Server Architecture lacks the robustness of Peer to Peer Architecture.

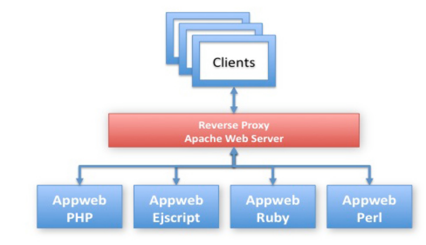
Lets look at this architecture implementation in ACME…



```
System simple_cs ={
   Component client ={
         Port send-request;
         Properties { Aesop-style : style-id = client-server;
                    UniCon-style : style-id = cs;
                    source-code:external = "CODE-LIB/client.c"}}
   Component server ={
         Port receive-request;
         Properties { idempotence : boolean = ture;
                    max-concurrent-clients : integer = 1;
                    source-code:external = "CODE-LIB/server.c"}}
   Connector rpc ={
         Roles {caller,callee}
         Properties { synchronous : boolean  =true
                    max-roles : integer = 2;
                    protocol : Wright = " "}}
   Attachments {
         client.send-request to rpc.caller;
         server.receive-request to rpc.callee}

}
```

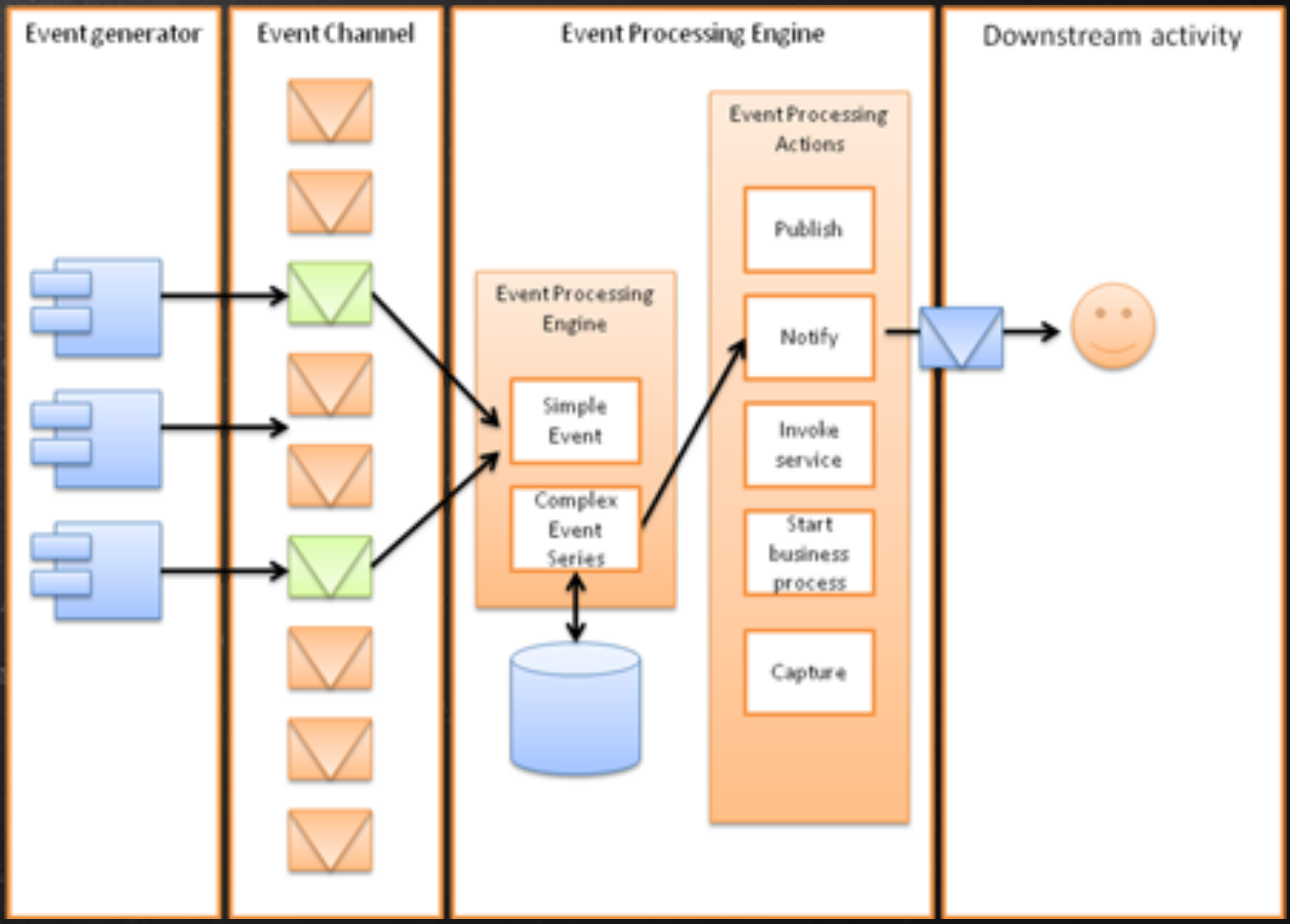
Client - Server in ACME

Event-Driven Architecture



- Architecture pattern that promotes production, detection, consumption of and reaction to events.
- It consists of event emitters and event consumers.
- Sinks have the responsibility of applying a reaction as soon as the event is presented.

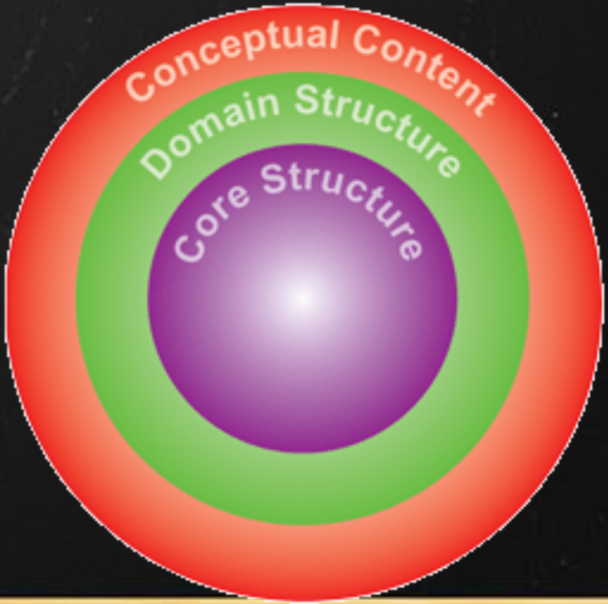
Systems have certain goal under the control of some message mechanism and the subsystem collaborates with each other to achieve system's ultimate goal.



Event - Driven Architecture

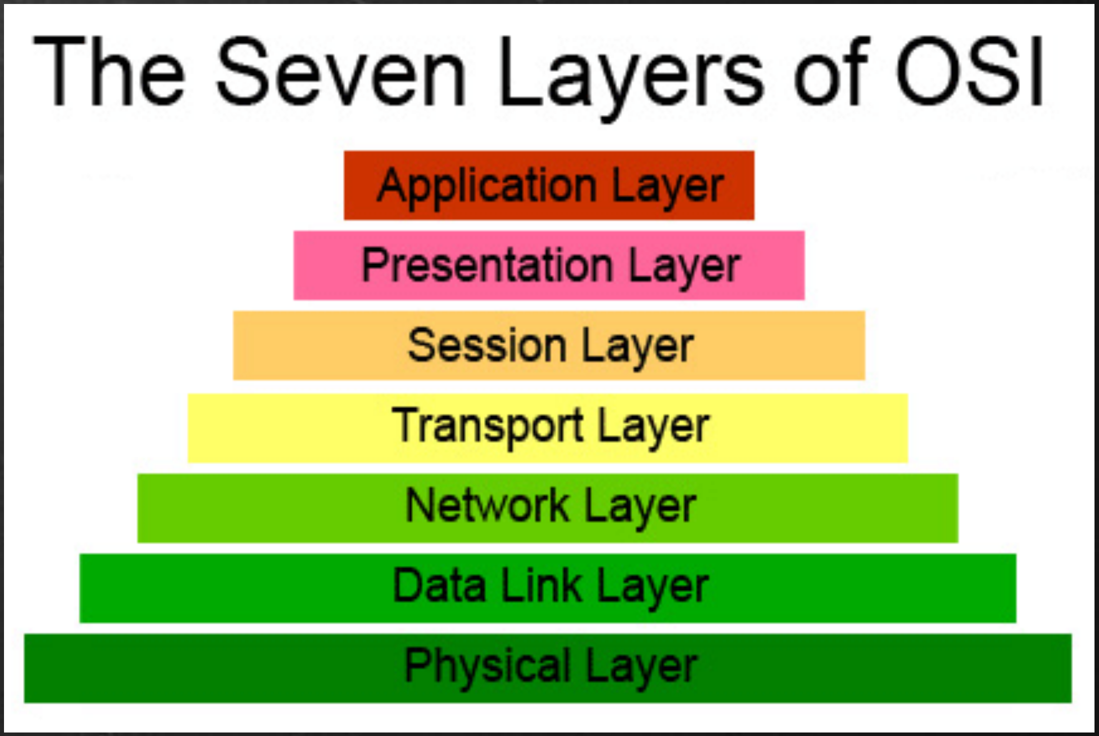
Hierarchical Layer

- It is a layered architecture.
- Each layer has 2 roles:
  1. Provide services for the upper layers.
  2. Call lower layers functions.
- Conceptual layer system model:



Advantages of Layering

- Supports gradual abstraction in the system design process.
- Layer system has good extendability.
- Layer style supports software reuse.



The Seven Layers of OSI

Application Layer

Presentation Layer

Session Layer

Transport Layer

Network Layer

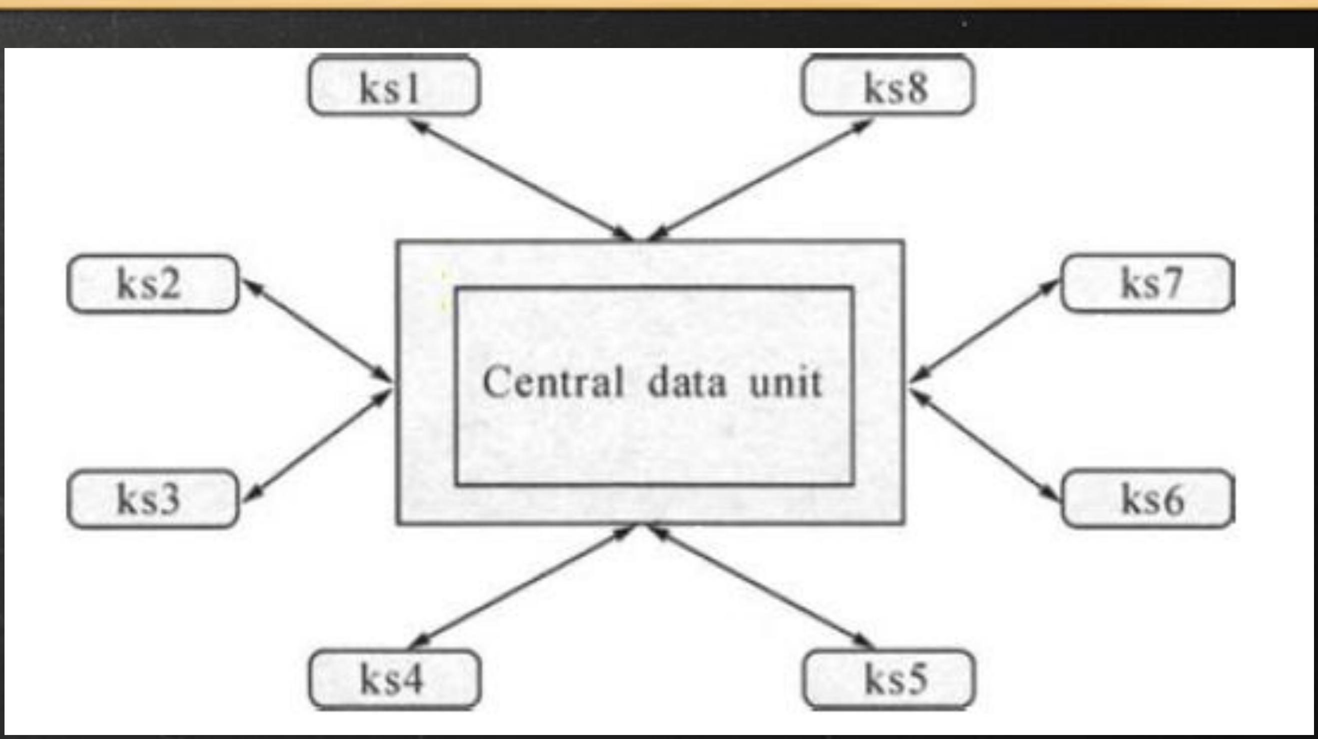
Data Link Layer

Physical Layer

Example of a layered architecture: ISO/OSI network 7- layer architecture

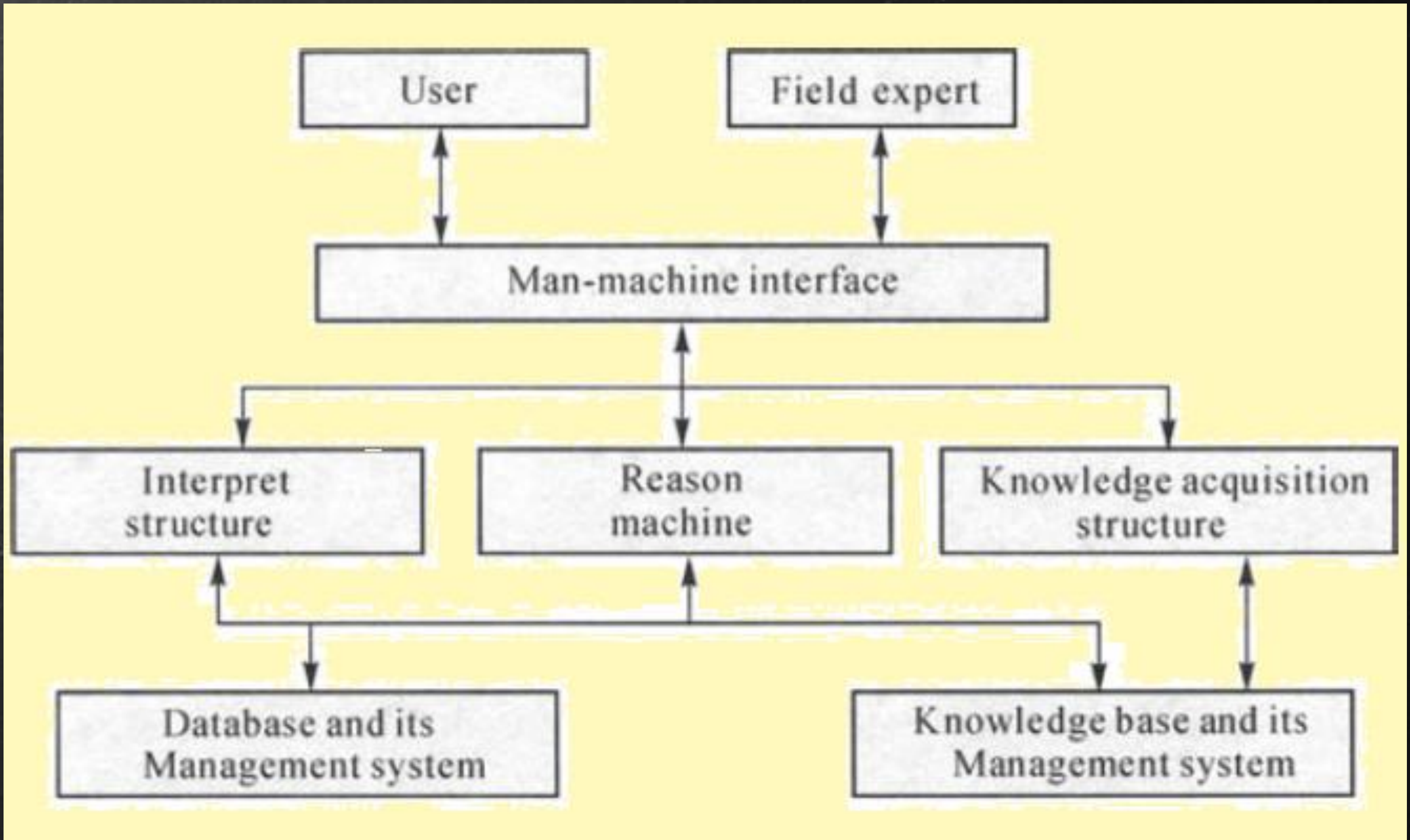
Data Sharing

- Also called repository style.
- System has 2 components:
  1. Central data unit component.
  2. Set of relatively dependent components.
- Central data unit called the repository shares information with all the other units.
- There are differences in the information exchange patterns.
- Thus there are 2 main control stratergies to deal with these information exchange patterns.



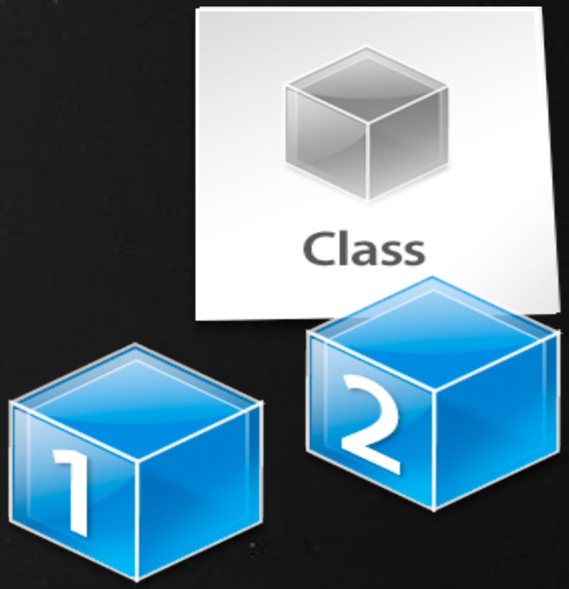
Black-board type repository model

The components:
ks-knowledge sources,
Central Data Unit,
Control Unit.

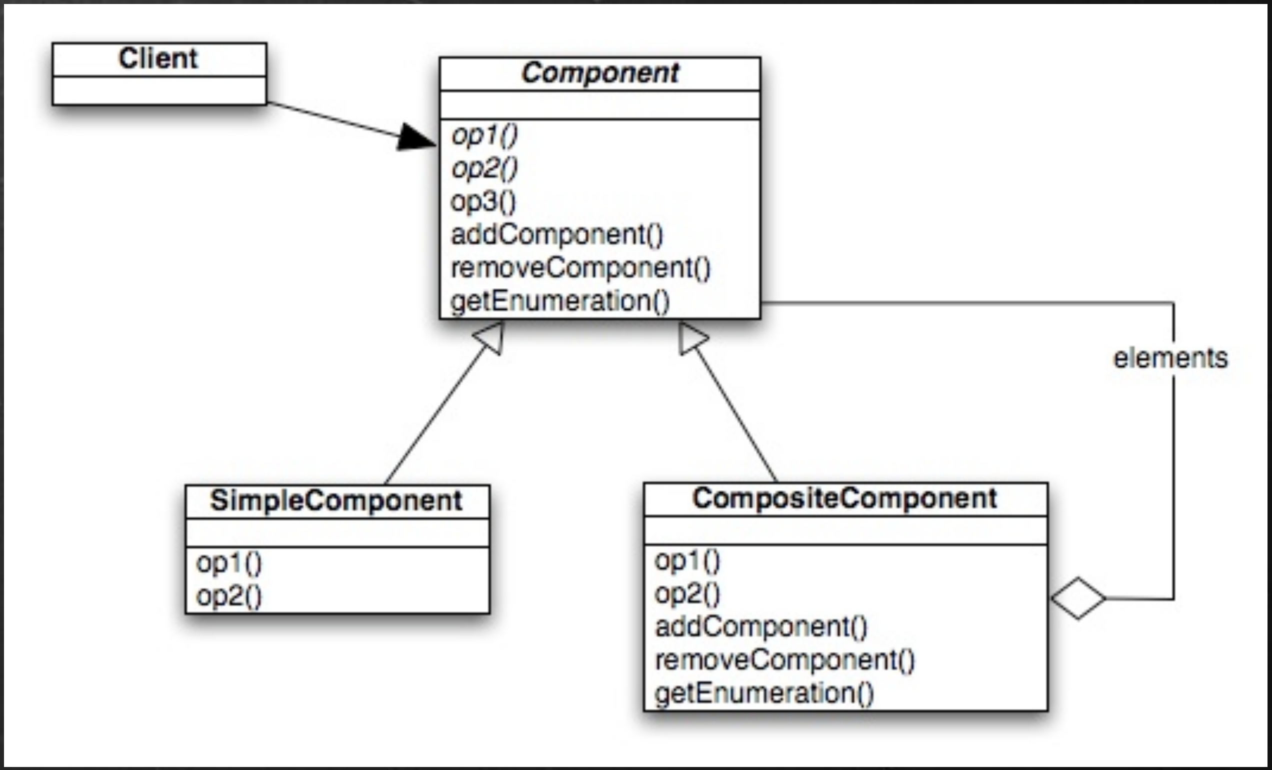

User
Field expert
Man-machine interface
Interpret structure
Reason machine
Knowledge acquisition structure
Database and its Management system
Knowledge base and its Management system

Example: Expert system

Object Oriented



- The key features are:
  - Data Abstraction.
  - Modularization.
  - Information encapsulation.
  - Inheritance.
  - Polymorphism.
- Objects in the problem are first recognized, then proper classes are constructed to represent these objects.
- Java - Object Oriented Programming, C - Procedural programming.



Example of Object Oriented Architecture:
Described using a UML diagram.

Architecture Description Languages

- Computer language used to describe the software architecture.
- Shaw and Garland's description for ADL's includes-
  1. Components.
  2. Operators.
  3. Patterns.
  4. Closure.
  5. Specification.
- Different ADL's existing: ACME, AADL, Darwin, WRIGHT.

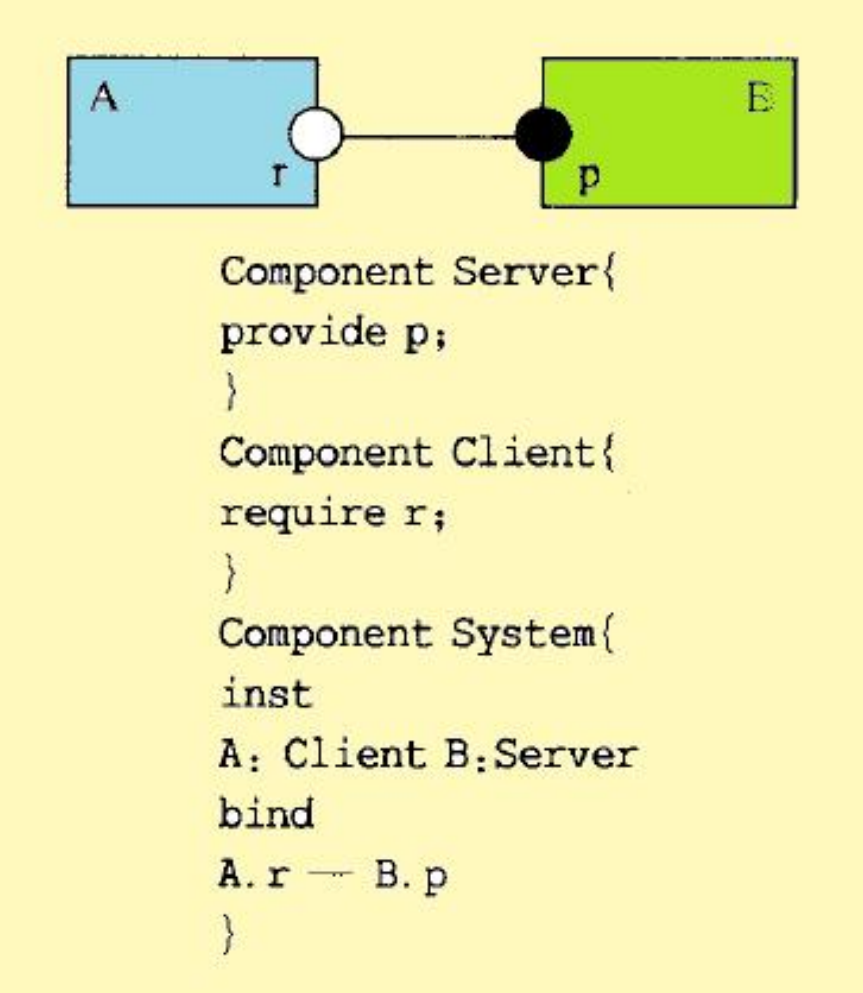
What makes a language an ADL?

- Be suitable for communicating an architecture to all the stake holders.
- Support the tasks of architecture creation, refinement and validation.
- Provide the ability to represent most common architectural styles.
- Support analytical capabilities.
- Provide quick generating prototype implementations.

Darwin

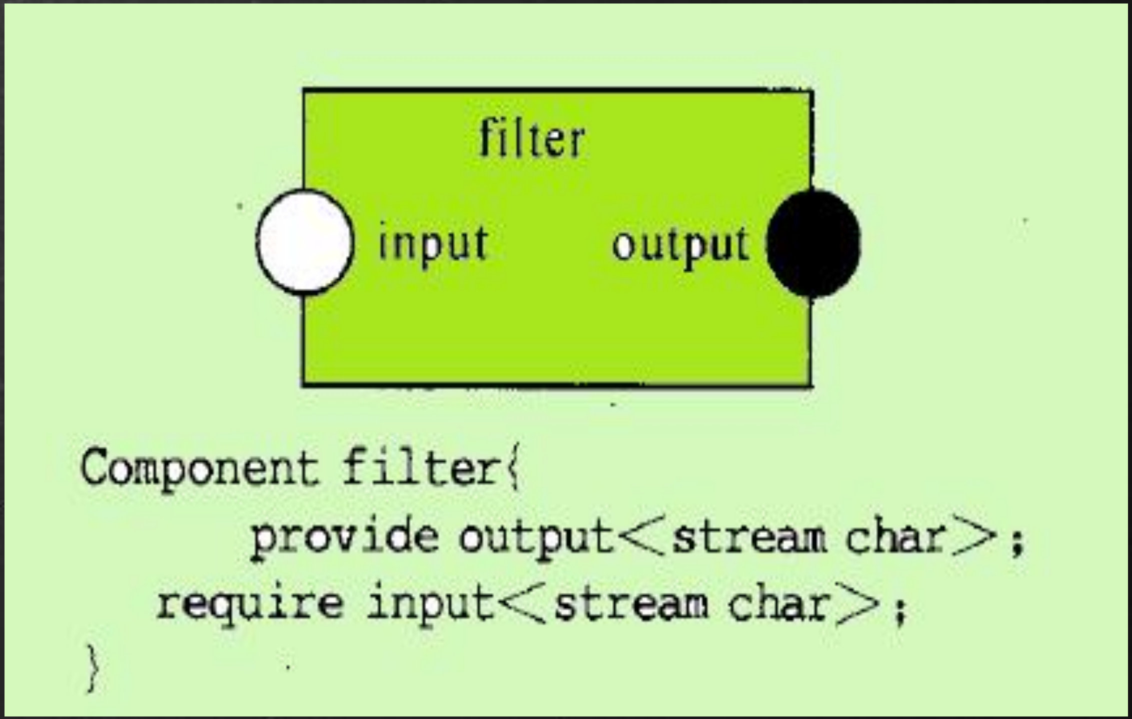
- Declarative Language.
- Describes the organization of software in terms of components, their interfaces and their binding components between them.
- Provides general purpose notations for specifying the structure of the system.
- Focuses on specification of distributed software system.
- Supports the specifications of dynamic structures.

TODAY BEING CHARLES DARWIN'S BIRTHDAY, LET'S BEGIN CLASS WITH A MOMENT OF SILENT EVOLVING.
SUNDAY SCHOOL
12 FEB



```
Component Server{
provide p;
}
Component Client{
require r;
}
Component System{
inst
A: Client B:Server
bind
A.r — B.p
}
```

Client Server System in Darwin



```
Component filter{
        provide output<stream char>;
    require input<stream char>;
}
```

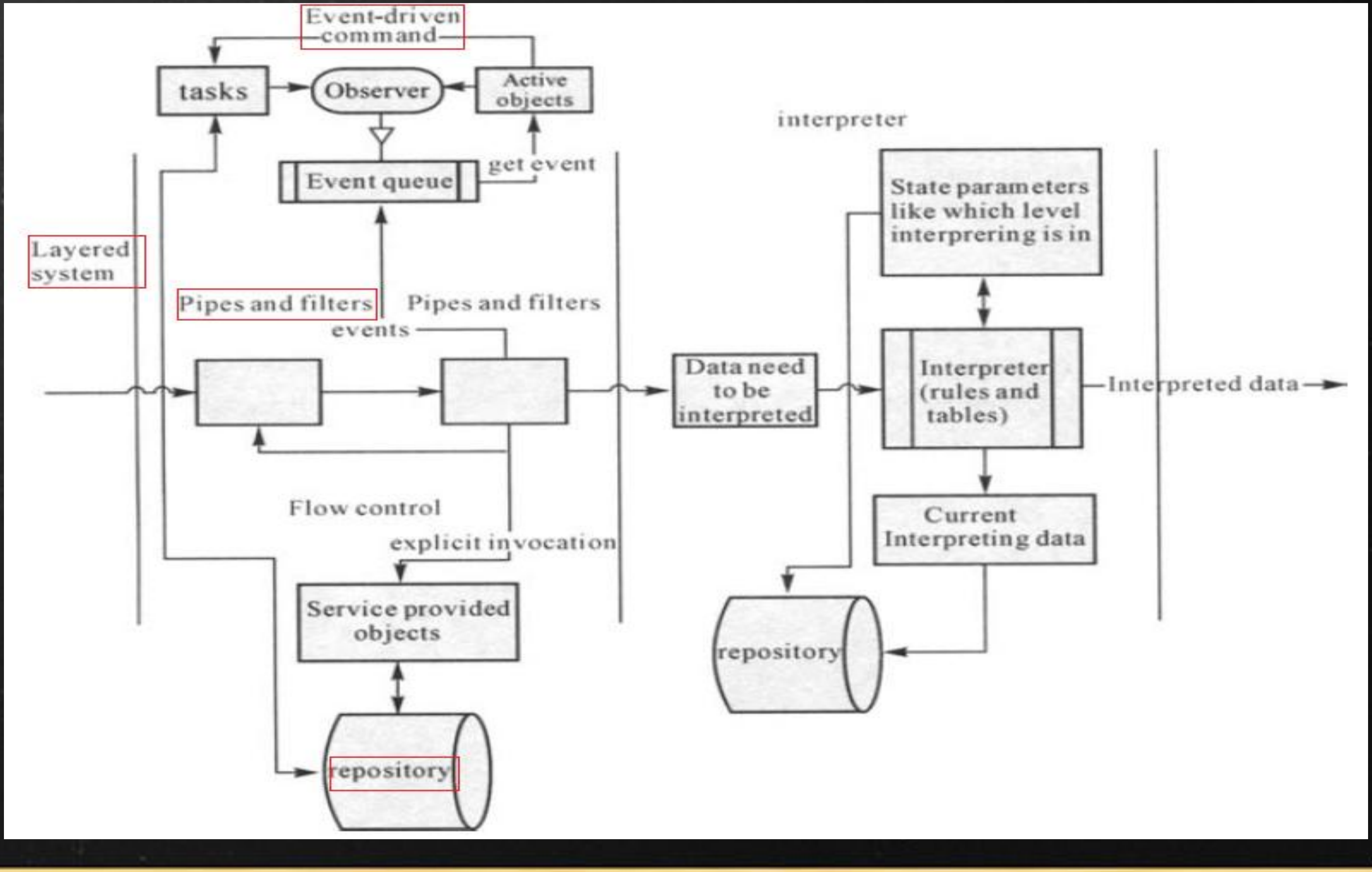
Filter Component in Darwin

Conclusion-I

- Common attribute in all the architectural slides - extendibility.
- Good software - closed for change, open for extension.
- Each style has its good quality attributes at the cost of sacrificing other quality attributes.
  - Pipes and filters style has bad interactivity while event driven style has good support for user interactivity.
  - In event driven style its hard to share common data, while repositories has advantage of data sharing.

Conclusion-II

- Maximum benefit of software architectural styles can be achieved by the integration of different styles.



References

- Software Architecture - Zheng Qin, Jiankuan Xing, Xiang Zheng.
- Garfixia Software Architecture - Patrick Van Bergen.
- Art of Software Architecture: Design methods and Techniques - S.T. Albin.

Software Engineering Institute

Software Architecture for Shared Information Systems

Mary Shaw

March 1993

TECHNICAL REPORT
CMU/SEI-93-TR-003

http://www.sei.cmu.edu

Carnegie Mellon

Technical Report
CMU/SEI-93-TR-3
ESC-TR-93-180
March 1993

Software Architecture for Shared Information Systems

Mary Shaw

School of Computer Science
and Software Engineering Institute

Software Engineering Information Modeling Project

This report will also appear as Carnegie Mellon University
School of Computer Science Technical Report No. CMU-CS-93-126.

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESC/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

The Software Engineering Institute is sponsored by the U.S. Department Defense.
This report was funded by the U.S. Department of Defense.

The work reported here was supported by the Carnegie Mellon University School of Computer Science and Software Engineering Institute (which is sponsored by the U.S. Department of Defense), by a grant from Siemens Corporate Research, and by the Department of Defense Advanced Research Project Agency under grant MDA972-92-J-1002. The views and conclusions are those of the author.

This document is available through the defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Service. For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161.

Copies of this document are also available from Research Access, Inc., 3400 Forbes Avenue, Suite 302, Pittsburgh, PA 15213.

Table of Contents

1. **Introduction** 1
   - 1.1 Systems Integration 2
   - 1.2 Shared Information Systems 3
   - 1.3 Design Levels 4
   - 1.4 External Software Systems 6
2. **Database Integration** 9
   - 2.1. Batch Sequential 9
   - 2.2. Simple Repository 12
   - 2.3. Virtual Repository 16
   - 2.4. Hierarchical Layers 18
   - 2.5. Evolution of Shared Information Systems in Business Data Processing 20
3. **Integration in Software Development Environments** 23
   - 3.1. Batch Sequential 23
   - 3.2. Transition from Batch Sequential to Repository 23
   - 3.3. Repository 26
   - 3.4. Hierarchical Layers 27
   - 3.5. Evolution of Shared Information Systems in Software Development Environments 29
4. **Integration in Building Design** 31
   - 4.1. Repository 32
   - 4.2. *Intelligent Control* 34
   - 4.3. Evolution of Shared Information Systems in Building Design 36
5. **Architectural Structures for Shared Information Systems** 37
   - 5.1 Variants on Data Flow Systems 37
   - 5.2. Variants on Repositories 38
6. **Conclusions** 41

Acknowledgments 43

References 45

Software Architectures for Shared Information Systems[1]

Abstract: Software system design takes place at many levels. Different kinds of design elements, notations, and analyses distinguish these levels. At the *software architecture* level, designers combine subsystems into complete systems. This paper studies some of the common patterns, or idioms, that guide these configurations. Results from software architecture offer some insight into the problems of systems integration—the task of connecting individual, isolated, pre-existing software systems to provide coherent, distributed solutions to large problems. As computing has become more sophisticated, so too have the software structures used in the integration task. This paper reviews historical examples of shared information systems in three different applications whose requirements share some common features about collecting, manipulating, and preserving large bodies of complex information. These applications have similar architectural histories in which a succession of designs responds to new technologies and new requirements for flexible, highly dynamic responses. A common pattern, the *shared information system evolution pattern,* appears in all three areas.

1. Introduction

Software system design takes place at many levels, each with its own concerns. We learn from computer hardware design that each of these levels has its own elements and composition operators and its own notations, analysis tools, and design rules. From the 1960s through the 1980s software developers concentrated on the programming level. At this level, so-called higher-level programming languages provide for the definitions of algorithms and data structures using the familiar programming language control statements, types, and procedures. Now we are turning our attention to the architectural level, in which patterns for organizing module-scale components guide software system design.

[1] To appear in *Mind Matters: Contributions to Cognitive and Computer Science in Honor of Allen Newell.* David Steier and Tom Mitchell (eds.), Hillsdale, N.J.: Lawrence Erlbaum Associates (to appear).

1.1 Systems Integration

Large software systems are often integrated from pre-existing systems. The designer of such a system must accommodate very different—often incompatible—conceptual models, representations, and protocols in order to create a coherent synthesis. Systems integration is a problem-solving activity that entails harnessing and coordinating the power and capabilities of information technology to meet a customer's needs. It develops megasystems in which pre-existing independent systems become subsystems—components that must interact with other components. Successful integration requires solution of both organizational and technical problems:

- understanding the current organizational capabilities and processes
- re-engineering and simplification of processes with a system view
- standardizing on common data languages and system architectures
- automation of processes and systems

Five kinds of issues motivate companies to invest in systems integration (CSTB 1992, pp. 16-21):

- For many organizations, experiences with information technology have not lived up to expectations.
- The proliferation of information technology products and vendors has produced the need for connectivity and interoperability.
- An installed base of information technology has to accommodate new technology and new capabilities.
- Advances in technology, combined with growing appreciation of what can be accomplished with that technology, have prompted firms to search for new applications and sources of competitive advantage.
- In an increasingly global economy, firms must rely on telecommunications and information technology to manage and coordinate their operations and to stay abreast of international competitors.

Corporate mergers and reorganizations, in particular, create needs for compatibility among systems developed under different assumptions about representation and interaction. The task is difficult: it involves large, untidy problems; incomplete, imprecise, and inconsistent requirements; and "legacy" systems that must be retained rather than replaced. For systems integration to be useful, it must be globally effective within the organization. The focus of this paper is on the enabling technologies rather than the organizational questions.

The essential enabling technologies are of several kinds (CSTB 1992, Nilsson et al 1990):

- *Architecture:* System organization; kinds of components, kinds of interactions, and patterns of overall organization; ability to evolve; consistency with available modular building blocks for hardware, software, and databases; standardization and open systems
- *Semantics:* Representations; conceptual consistency; semantic models; means for handing inconsistencies
- *Connectivity:* Mechanisms for moving data between systems and initiating action in other systems; communication platforms with flexible linkages or interfaces; network management and reliability; security
- *Interaction:* Granularity; user interfaces; interoperability; simplicity; underlying consistency of presentation

The technologies for architecture are of primary interest here; to a certain extent these are inseparable from semantics.

1.2 Shared Information Systems

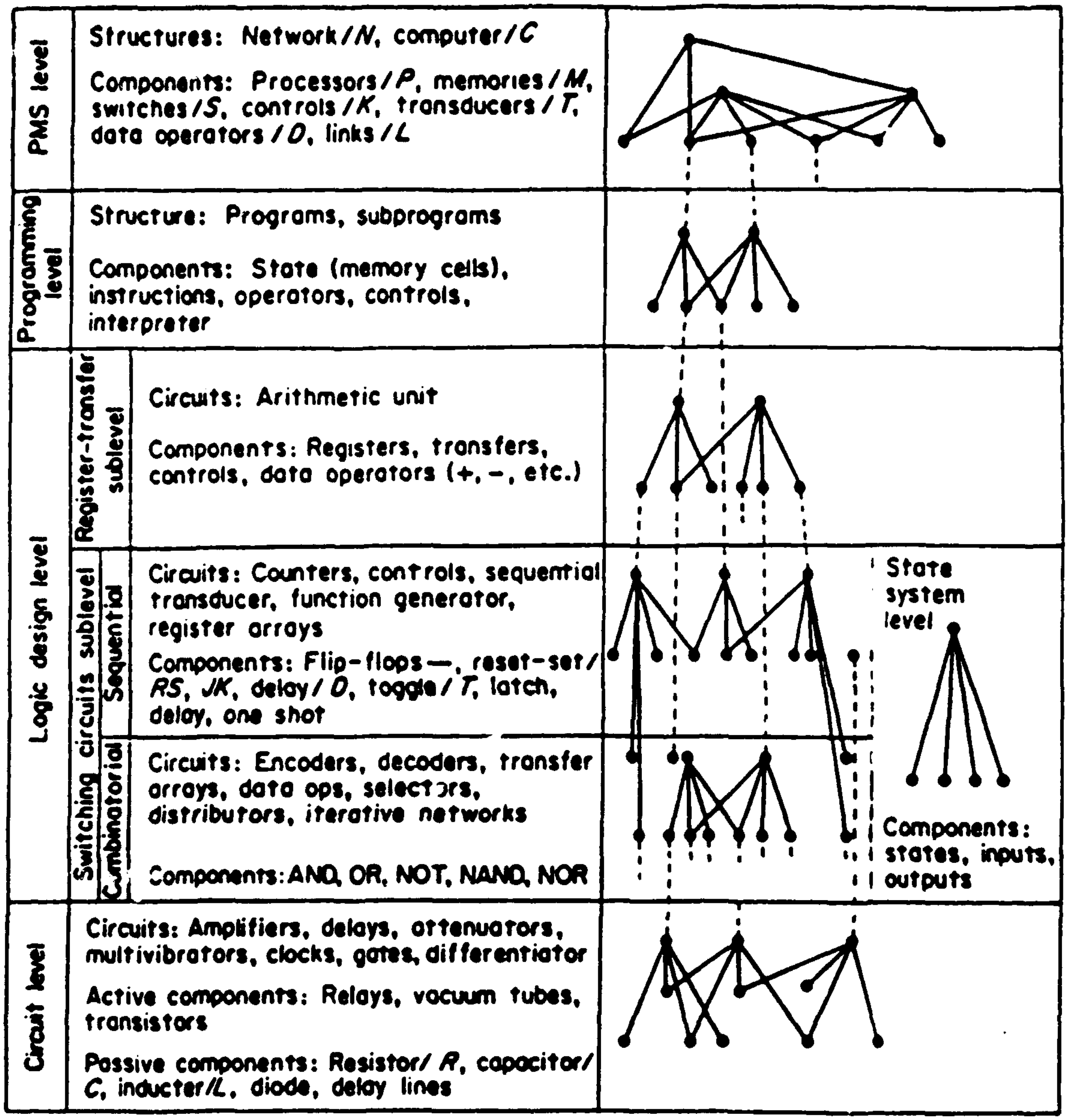
One particularly significant class of large systems is responsible for collecting, manipulating, and preserving large bodies of complex information. These are *shared information systems.* Systems of this kind appear in many different domains; this paper examines three. The earliest shared information systems consisted of separate programs for separate subtasks. Later, multiple independent processing steps were composed into larger tasks by passing data in a known, fixed format from one step to another. This organization is not flexible in responding to variations or discrepancies in data. Nor is it tolerant of structural modification, especially the addition of components developed under different assumptions. It is also not responsive to the needs of interactive processing, which must handle individual requests as they arrive.

Still later, often when requirements for interaction appear, new organizations allowed independent processing subsystems to interact through a shared data store. While this organization is an improvement, it still encounters integration problems—especially when multiple data stores with different representations must be shared, when the system is distributed, when many user tasks must be served, and when the suite of processing and data subsystems changes regularly. Several newer approaches now compensate for these differences in representation and operating assumptions, but the problem is not completely solved. A common pattern, the *shared information system evolution pattern,* is evident in the application areas examined here.

1.3 Design Levels

System design takes place at many levels. It is useful to make precise distinctions among those levels, for each level appropriately deals with different design concerns. At each level we find *components,* both primitive and composite; *rules of composition* that allow the construction of nonprimitive components, or systems; and *rules of behavior* that provide semantics for the system (Bell and Newell 1971, Newell 1982, Newell 1990). Since these differ from one level to another, we also find different notations, design problems, and analysis techniques at each level. As a result, design at one level can proceed substantially autonomously of any other level. But levels are also related, in that elements at the base of each level correspond to—are implemented by—constructs of the level below.

The hierarchy of levels for computer hardware systems is familiar and appears in Figure 1 (Bell and Newell 1971, p. 3). Note first that each level deals with different content. Different kinds of structures guide design with different sets of components. Different notations, analysis techniques, and design issues accompany the differences of content matter. Note also that each level admits of substructure: abstraction and composition take place within each level, in terms of the components and structures of that level. In addition, there is an established transformation from the primitive components at the bottom of each level to (probably nonprimitive) components of the level below.



PMS level			Structures: Network/*N*, computer/*C* Components: Processors/*P*, memories/*M*, switches/*S*, controls/*K*, transducers/*T*, data operators/*D*, links/*L*
Programming level			Structure: Programs, subprograms Components: State (memory cells), instructions, operators, controls, interpreter
Logic design level	Register-transfer sublevel		Circuits: Arithmetic unit Components: Registers, transfers, controls, data operators (+, −, etc.)
	Switching circuits sublevel	Sequential	Circuits: Counters, controls, sequential transducer, function generator, register arrays Components: Flip-flops —, reset-set/*RS*, *JK*, delay/*D*, toggle/*T*, latch, delay, one shot
		Combinatorial	Circuits: Encoders, decoders, transfer arrays, data ops, selectors, distributors, iterative networks Components: AND, OR, NOT, NAND, NOR
Circuit level			Circuits: Amplifiers, delays, attenuators, multivibrators, clocks, gates, differentiator Active components: Relays, vacuum tubes, transistors Passive components: Resistor/*R*, capacitor/*C*, inducter/*L*, diode, delay lines

Figure 1: Computer hardware design levels

Software, too, has its design levels. We can identify at least

- *Architecture*, where the design issues involve overall association of system capability with components; components are modules, and interconnections among modules are handled in a variety of ways, all of which seem to be expressed as explicit sets of procedure calls.
- *Code*, where the design issues involve algorithms and data structures; the components are programming language primitives such as numbers, characters, pointers, and threads of control; primitive operators are the arithmetic and data manipulation primitives of the language; composition mechanisms include records, arrays, and procedures.
- *Executable*, where the design issues involve memory maps, data layouts, call stacks, and register allocations; the components are bit patterns supported by hardware, and the operations and compositions are described in machine code.

These roughly track the higher levels of hardware design. The executable and code levels for software are well understood. However, the architecture level is currently understood mostly at the level of intuition, anecdote, and folklore. It is common for a description of a software system to include a few paragraphs of text and a box-and-line diagram, but there is neither uniform syntax nor uniform semantics for interpreting the prose and the diagrams. Our concern here is in improving understanding and precision at the *software architecture level*. At this level the components are programs, modules, or systems; a rich collection of interchange representations and protocols connect the components; and well-known system patterns guide the compositions (Garlan and Shaw 1993).

1.4 External Software Systems

Recent work on intelligent integration of *external software systems* offers some hope for improving the sophistication of our integration techniques. Newell and Steier (1991) suggest that the work on agent-ESS systems may contribute to software engineering by making the power of computer software more easily accessible in the service of computational tasks. An intelligent system would learn to recognize aberrations when they arise and compensate for them, and it would adapt to new protocols and representations when the suite of available components changes.

This paper explores what happens when independent systems become components of larger systems. It examines three examples of shared information systems:

- *Data processing,* driven primarily by the need to build business decision systems from conventional databases
- *Software development environments,* driven primarily by the need to represent and manipulate programs and their designs.
- *Building design,* driven primarily by the need to couple independent design tools to allow for the interactions of their results in structural design

We will see how the software architectures of these systems changed as technology and demands on system performance changed. We close by surveying the architectural constructs used to describe the examples and examining the prospects for intelligent integration.

2. Database Integration

Business data processing has traditionally been dominated by database management, in particular by database updates. Originally, separate databases served separate purposes, and implementation issues revolved around efficient ways to do massive coordinated periodic updates. As time passed, interactive demands required individual transactions to complete in real time. Still later, as databases proliferated and organizations merged, information proved to have value far beyond its original needs. Diversity in representations and interfaces arose, information began to appear redundantly in multiple databases, and geographic distribution added communication complexity. As this happened, the challenges shifted from individual transaction processing to integration.

Individual database systems must support transactions of predetermined types and periodic summary reports. Bad requests require a great deal of special handling. Originally the updates and summary reports were collected into batches, with database updates and reports produced during periodic batch runs. However, when interactive queries became technologically possible, the demand for interaction made generated demand for on-line processing of both individual requests and exceptions. Reports remained on roughly the same cycles as before, so reporting became decoupled from transaction processing.

As databases became more common, information about a business became distributed among multiple databases. This offered new opportunities for the data to become inconsistent and incomplete. In addition, the representations, or schemas, for different databases were usually different; even the portion of the data shared by two databases is likely to have representations in each database. The total volume of data to handle is correspondingly larger, and it is often distributed across multiple machines. Two general strategies emerged for dealing with data diversity: unified schemas and multi-databases.

2.1. Batch Sequential

Some of the earliest large computer applications were databases. In these applications individual database operations—transactions—were collected into large batches. The application consisted of a small number of large standalone programs that performed sequential updates on flat (unstructured) files. A typical organization included:

- a massive *edit program*, which accepted transaction inputs and performed such validation as was possible without access to the database

- a massive *transaction sort*, which got the transactions into the same order as the records on the sequential master file
- a sequence of *update programs*, one for each master file; these huge programs actually executed the transactions by moving sequentially through the master file, matching each type of transaction to its corresponding account and updating the account records
- a *print program* that produced periodic reports

The steps were independent of each other; they had to run in a fixed sequence; each ran to completion, producing an output file in a new format, before the next step began. This is a *batch sequential* architecture. The organization of a typical batch sequential update system appears in Figure 2 (Best 1990, p. 29). This figure also shows the possibility of on-line queries (but not modifications). In this structure the files to support the queries are reloaded periodically, so recent transactions (e.g., within the past few days) are not reflected in the query responses.

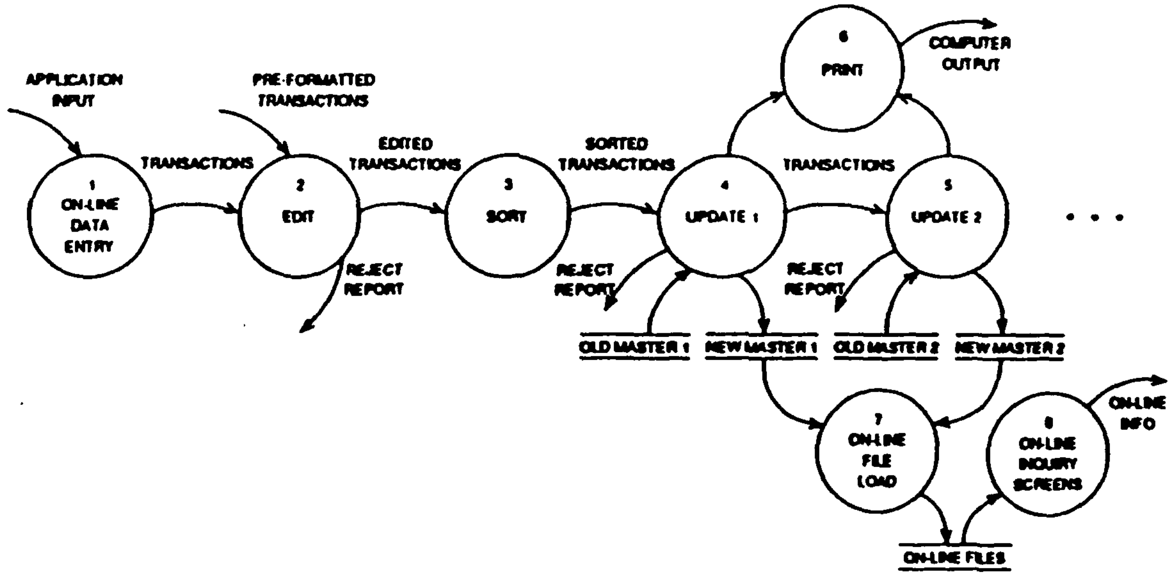
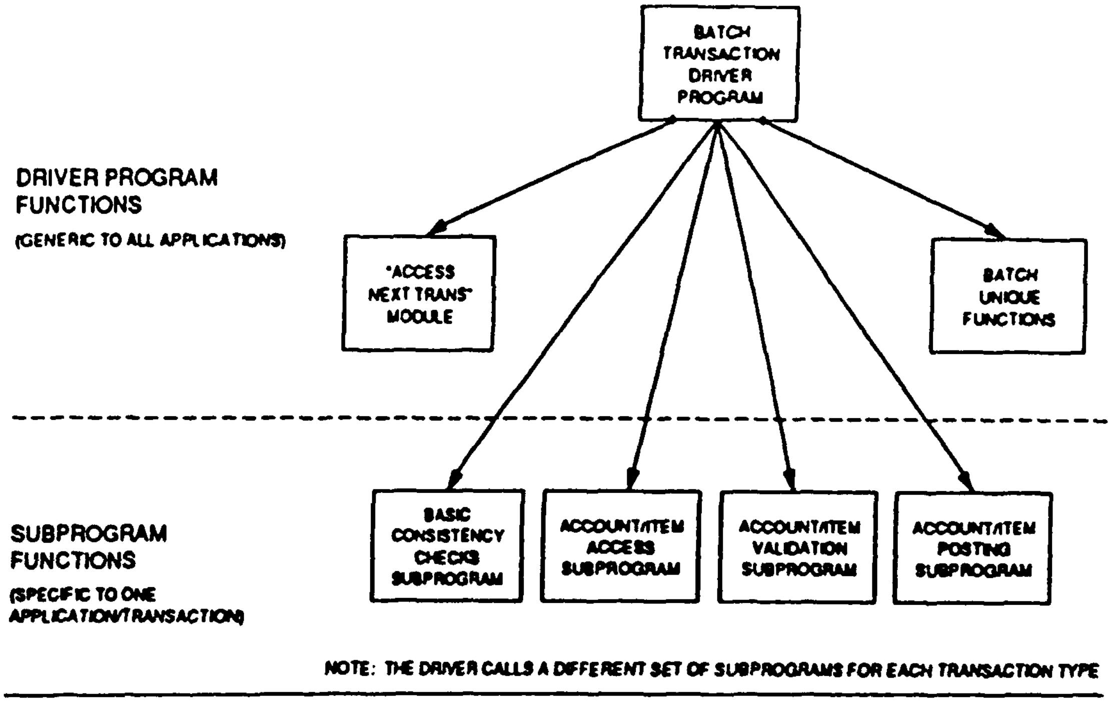


Figure 2: Data flow diagram for batch databases

Figure 2 is a Yourdon data flow diagram. Processes are depicted as circles, or "bubbles"; data flow (here, large files) is depicted with arrows, and data stores such as computer files are depicted with parallel lines. This notation conventional in this application area for showing the relations among processes and data flow. Within a bubble, however, the approach changes. Figure 3 (Best 1990, p.150) shows the internal structure of an update process. There is one of these for each of the master data files, and each is responsible for handling all possible updates to that data file.


BATCH TRANSACTION DRIVER PROGRAM
DRIVER PROGRAM FUNCTIONS
(GENERIC TO ALL APPLICATIONS)
"ACCESS NEXT TRANS" MODULE
BATCH UNIQUE FUNCTIONS
SUBPROGRAM FUNCTIONS
(SPECIFIC TO ONE APPLICATION/TRANSACTION)
BASIC CONSISTENCY CHECKS SUBPROGRAM
ACCOUNT/ITEM ACCESS SUBPROGRAM
ACCOUNT/ITEM VALIDATION SUBPROGRAM
ACCOUNT/ITEM POSTING SUBPROGRAM

NOTE: THE DRIVER CALLS A DIFFERENT SET OF SUBPROGRAMS FOR EACH TRANSACTION TYPE

Figure 3: Internal structure of batch update process

In Figure 3, the boxes represent subprograms and the lines represent procedure calls. A single driver program processes all batch transactions. Each transaction has a standard set of subprograms that check the transaction request, access the required data, validate the trεnsaction, and post the result. Thus all the program logic for each transaction is localized in a single set of subprograms. The figure indicates that the transaction-processing template is replicated so that each transaction has its own set. Note the difference even in graphical notation as the design focus shifts from the architecture to the code level.

The essential—batch sequential—parts of Figure 2 are redrawn in Figure 4 in a form that allows comparison to other architectures. The redrawn figure emphasizes the sequence of operations to be performed and the completion of each step before the start of its successor. It suppresses the on-line query support and updates to multiple master files, or databases.

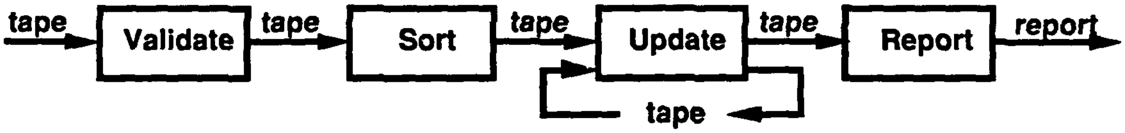


Figure 4: Batch sequential database architecture

2.2. Simple Repository

Two trends forced a change away from batch sequential processing. First, interactive technology provided the opportunity and demand for continuous processing of on-line updates as well as on-line queries. On-line queries of stale data are not very satisfactory; interaction requires incremental updates of the database, at least for on-line transactions (there is less urgency about transactions that arrive by slow means such as mail, since they have already incurred delays). Second, as organizations grew, the set of transactions and queries grew. Modifying a single large update program and a single large reporting program for each change to a transaction creates methodological bottlenecks. New types of processing were added often enough to discourage modification of a large update program for each new processing request. In addition, starting up large programs incurred substantial overheads at that time.

These trends led to a change in system organization. Figure 5 (Best 1990, p. 81) shows a "modern"—that is, interactive—system organization. The notation is as for Figure 2. This organization supports both interactive and batch processing for all transaction types; updates can occur continuously. Since these are no longer periodic operations, the system also provides for periodic operations. Here, though, the transaction database and extract database are transient buffers; the account/item database is the central permanent store. The transaction database serves to synchronize multiple updates. The extract database solves a problem created by the addition of interactive processing—namely the loss of synchronization between the updating and reporting cycles. This figure obscures not only the difference between a significant database and a transient buffer but also the separation of transactions into separate processes.

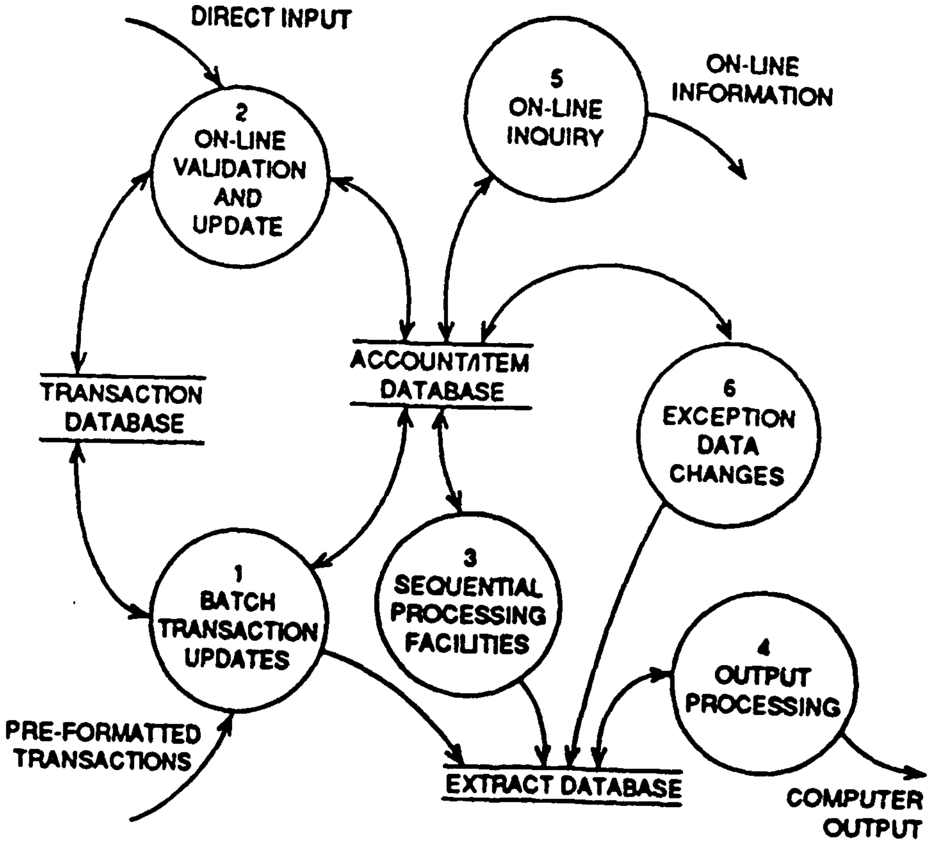


Figure 5: Dataflow diagram for interactive database

It is possible for transaction processing in this organization to resemble batch sequential processing. However, it is useful to separate the general overhead operations from the transaction-specific operations. It may also be useful to perform multiple operations on a single account all at once. Figure 6 (Best 1990, p.158) shows the program structure for the transactions in this new architecture. Since the transactions now exist individually rather than as alternatives within a single program, several of the bubbles in Figure 5 actually represent sets of independent bubbles.

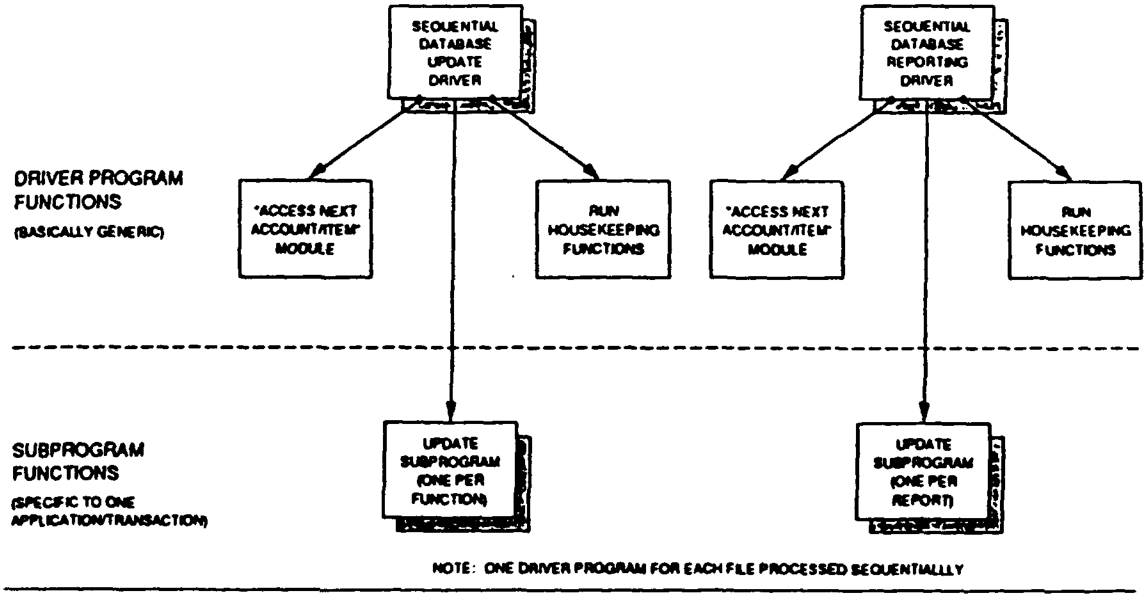

SEQUENTIAL DATABASE UPDATE DRIVER
SEQUENTIAL DATABASE REPORTING DRIVER
DRIVER PROGRAM FUNCTIONS
(BASICALLY GENERIC)
"ACCESS NEXT ACCOUNT/ITEM" MODULE
RUN HOUSEKEEPING FUNCTIONS
"ACCESS NEXT ACCOUNT/ITEM" MODULE
RUN HOUSEKEEPING FUNCTIONS
SUBPROGRAM FUNCTIONS
(SPECIFIC TO ONE APPLICATION/TRANSACTION)
UPDATE SUBPROGRAM (ONE PER FUNCTION)
UPDATE SUBPROGRAM (ONE PER REPORT)
NOTE: ONE DRIVER PROGRAM FOR EACH FILE PROCESSED SEQUENTIALLLY

Figure 6: Internal structure of interactive update process

There is not a clean separation of architectural issues from coding issues in Figures 5 and 6. It is not unusual to find this, because explicit attention to the architecture as a separate level of software design is relatively recent. Indeed, Figures 5 and 6 suffer from information overload as well. The system structure is easier to understand if we first isolate the database updates. Figure 7 focuses narrowly on the database and its transactions. This is an instance of a fairly common architecture, a *repository*, in which shared persistent data is manipulated by independent functions each of which has essentially no permanent state. It is the core of a database system. Figure 8 adds two additional structures. The first is a control element that accepts the batch or interactive stream of transactions, synchronizes them, and selects which update or query operations to invoke, and in which order. This subsumes the transaction database of Figure 5. The second is a buffer that serves the periodic reporting function. This subsumes the extract database of Figure 5.

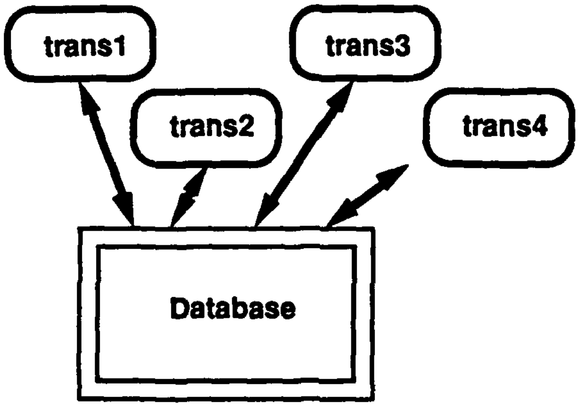


Figure 7: Simple repository database architecture

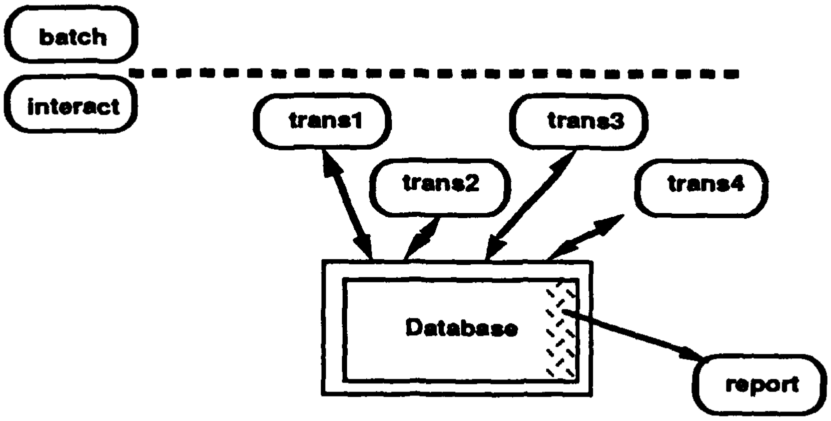
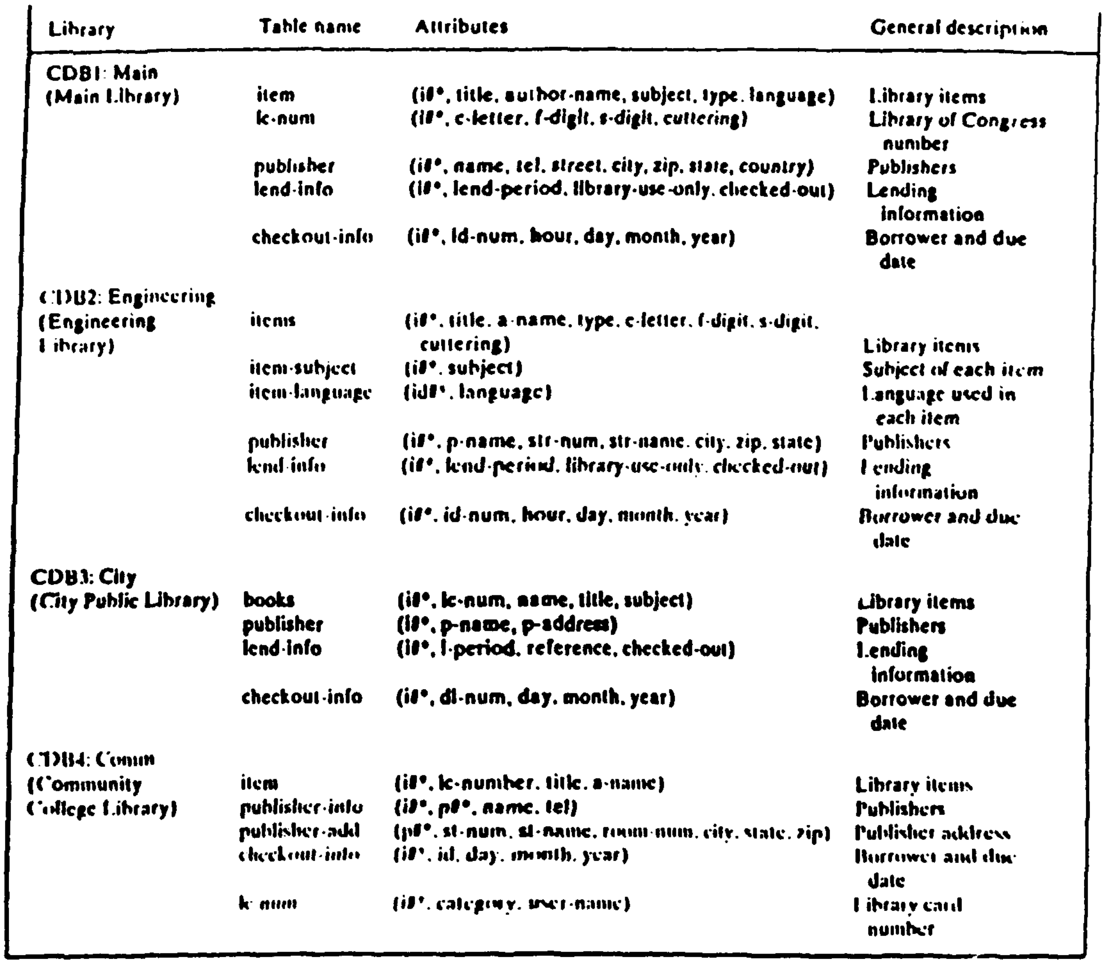


Figure 8: Repository architecture for database showing control and reporting

2.3. Virtual Repository

As organizations grew, databases came to serve multiple functions. Since this was usually a result of incremental growth, individual independent programs continued to he the locus of processing. In response, simple repositories gave way to databases that supported multiple views through schemas. Corporate reorganizations, mergers, and other consolidations of data forced the joint use of multiple databases. As a result, information could no longer be localized in a single database. Figure 9 (Kim and Seo 1991, p.13) gives a hint of the extent of the problem through the schemas that describe books in four libraries. Note, for example, that the call number is represented in different ways in all four schemas; in this case they're all Library of Congress numbers, so the more difficult case of a mixture of Library of Congress and Dewey numbering doesn't arise. Note also the assortment of ways the publisher's name, address, and (perhaps) telephone number are represented.



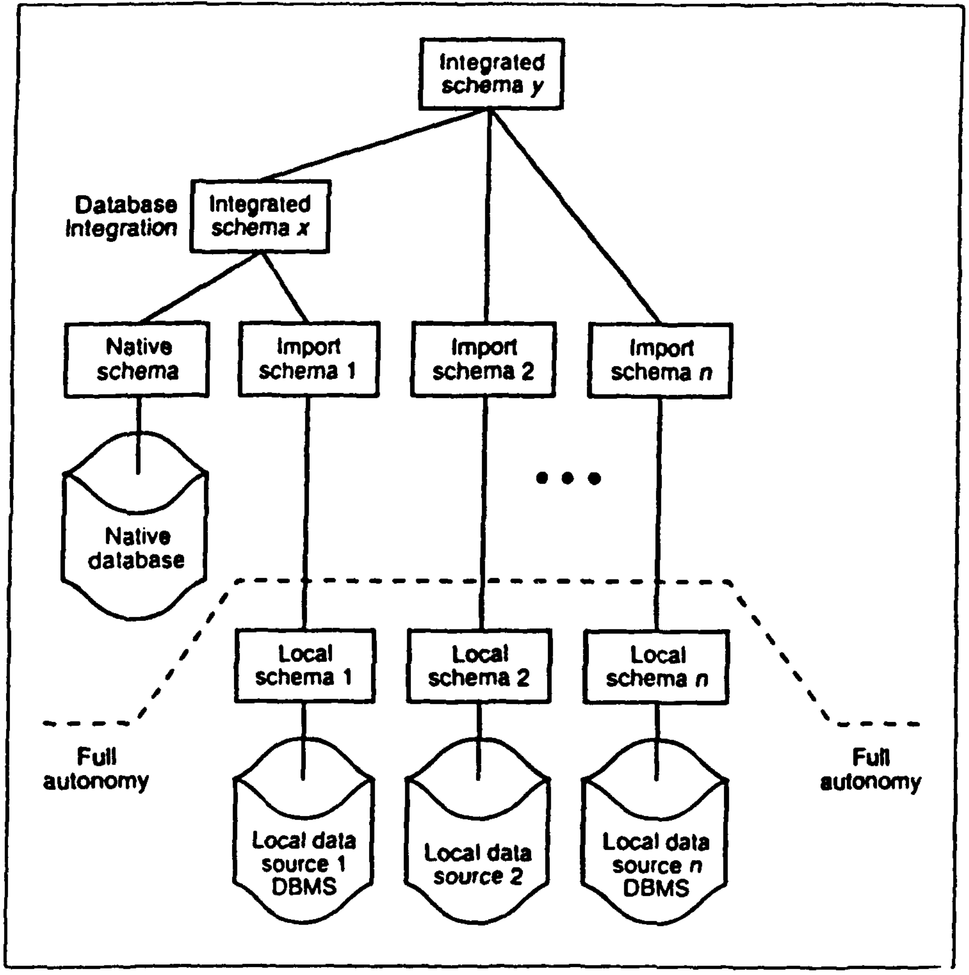
Library	Table name	Attributes	General description
CDB1: Main (Main Library)	item	(i#*, title, author-name, subject, type, language)	Library items
	lc-num	(i#*, c-letter, f-digit, s-digit, cuttering)	Library of Congress number
	publisher	(i#*, name, tel, street, city, zip, state, country)	Publishers
	lend-info	(i#*, lend-period, library-use-only, checked-out)	Lending information
	checkout-info	(i#*, id-num, hour, day, month, year)	Borrower and due date
CDB2: Engineering (Engineering Library)	items	(i#*, title, a-name, type, c-letter, f-digit, s-digit, cuttering)	Library items
	item-subject	(i#*, subject)	Subject of each item
	item-language	(id#*, language)	Language used in each item
	publisher	(i#*, p-name, str-num, str-name, city, zip, state)	Publishers
	lend-info	(i#*, lend-period, library-use-only, checked-out)	Lending information
	checkout-info	(i#*, id-num, hour, day, month, year)	Borrower and due date
CDB3: City (City Public Library)	books	(i#*, lc-num, name, title, subject)	Library items
	publisher	(i#*, p-name, p-address)	Publishers
	lend-info	(i#*, l-period, reference, checked-out)	Lending information
	checkout-info	(i#*, dl-num, day, month, year)	Borrower and due date
CDB4: Comm (Community College Library)	item	(i#*, lc-number, title, a-name)	Library items
	publisher-info	(i#*, p#*, name, tel)	Publishers
	publisher-add	(p#*, st-num, st-name, room-num, city, state, zip)	Publisher address
	checkout-info	(i#*, id, day, month, year)	Borrower and due date
	lc-num	(i#*, category, user-name)	Library card number

*Indicates key attribute

© 1991 IEEE

Figure 9: Diversity of schemas for a single construct

Developing applications that rely on multiple diverse databases like these requires solution of two problems. First, the system must reconcile representation differences. Second, it must communicate results across distributed systems that may have not only different data representations but also different database schema representations. One approach to the unification of multiple schemas is called the federated approach. Figure 10 (Ahmed et al 1991, p. 21) shows one way to approach this, relying on the well-understood technology for handling multiple views on databases. The top of this figure shows how the usual database mechanisms integrate multiple schemas into a single schema. The bottom of the figure suggests an approach to importing data from autonomous external databases: For each database, devise a schema in its native schema language that exports the desired data and a matching schema in the schema language of the importer. This separates the solutions to the two essential problems and restricts the distributed system problem to communication between matching schemas.



© 1991 IEEE

Figure 10: Combining multiple distributed schemas

Figure 10 combines solutions to two problems. Here again, the design is clearer if the discussion and diagram separate the two sets of concerns. Figure 11 shows the integration of multiple databases by unified schemas. It shows a simple composition of projections. The details about whether the data paths are local or distributed and whether the local schema and import schema are distinct are suppressed at this level of abstraction; these communication questions should be addressed in an expansion of the abstract filter design (and they may not need to be the same for all of the constituents).

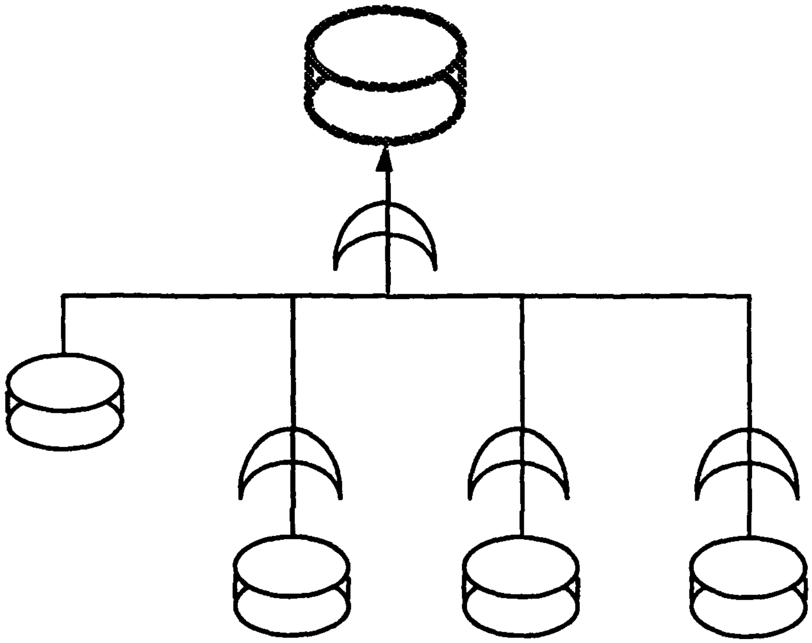
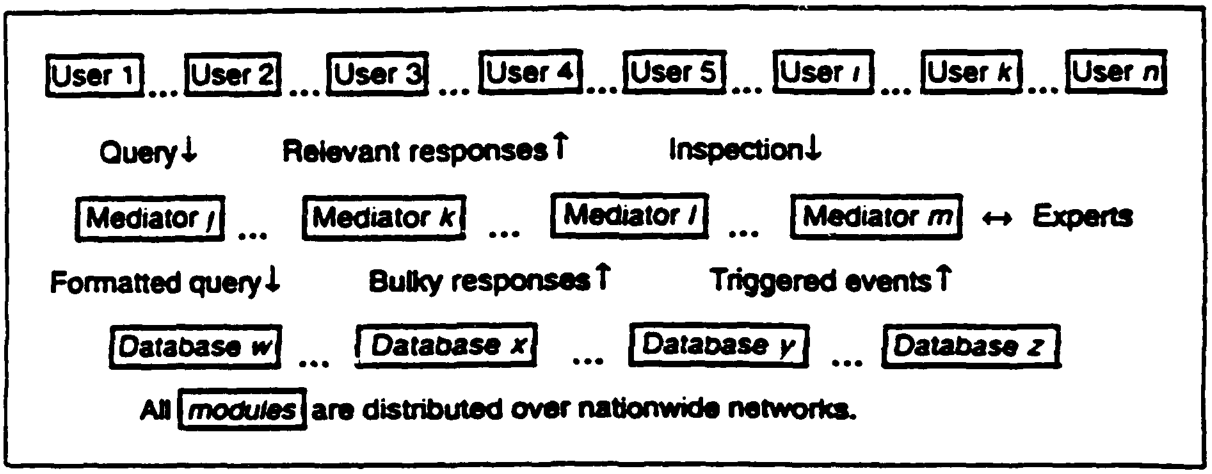


Figure 11: Integration of multiple databases

2.4. Hierarchical Layers

Unified schemas allow for merger of information, but their mappings are fixed, passive, and static. The designers of the views must anticipate all future needs; the mappings simply transform the underlying data; and there are essentially no provisions for recognizing and adapting to changes in the set of available databases. In the real world, each database serves multiple users, and indeed the set of users changes regularly. The set of available databases also changes, both because the population of databases itself changes and because network connectivity changes the set that is accessible. This exacerbates the usual problems of inconsistency across a set of databases. The commercial database community has begun to respond to this problem of dynamic reconfiguration. Distributed database products organized on a client-server model are beginning to challenge traditional mainframe database management systems (Hovaness 1992). This set of problems is also of current interest in the database research community.

Figure 12 (Wiederhold 1992, p. 45) depicts one research scenario for active mediation between a constantly-changing set of users and a constantly-changing set of databases. Wiederhold proposes introducing active programs, called experts, to accept queries from users, recast them as queries to the available databases, and deliver appropriate responses to the users. These experts, or active mediators, localize knowledge about how to discover what databases are available and interact with them, about how to recast users' queries in useful forms, and about how to reconcile, integrate, and interpret information from multiple diverse databases.


User 1 ... User 2 ... User 3 ... User 4 ... User 5 ... User *i* ... User *k* ... User *n*
Query↓
Relevant responses↑
Inspection↓
Mediator *j* ... Mediator *k* ... Mediator *l* ... Mediator *m* ↔ Experts
Formatted query↓
Bulky responses↑
Triggered events↑
Database *w* ... Database *x* ... Database *y* ... Database *z*
All *modules* are distributed over nationwide networks.

© 1992 IEEE

Figure 12: Multidatabase with mediators

In effect, Wiederhold's architecture uses *hierarchical layers* to separate the business of the users, the databases, and the mediators. The interaction between layers of the hierarchy will most likely be a *client-server* relation. This is not a repository because there is no enforced coherence of central shared data; it is not a batch sequential system (or any other form of pipeline) because the interaction with the data is incremental. Figure 13 recasts this in a form similar to the other examples.

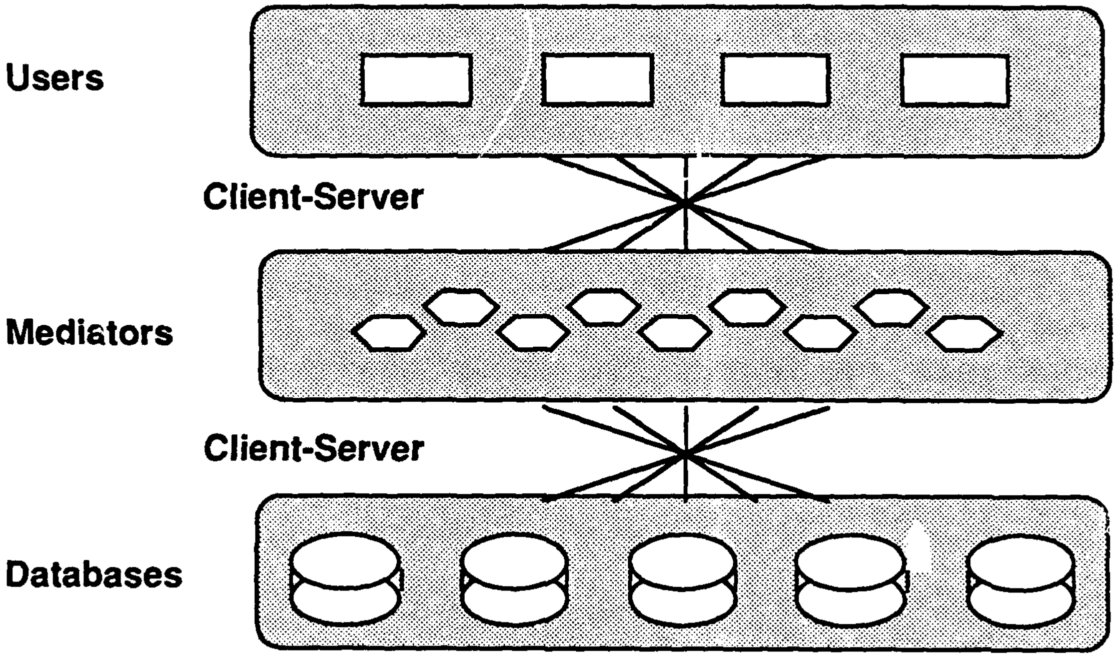


Figure 13: Layered architecture for multidatabase

2.5. Evolution of Shared Information Systems in Business Data Processing

These business data processing applications exhibit a pattern of development driven by changing technology and changing needs. The pattern was:

- *Batch processing:* Standalone programs; results are passed from one to another on magtape. *Batch sequential model.*
- *Interactive processing:* Concurrent operation and faster updates preclude batching, so updates are out of synchronization with reports. *Repository model* with external control.
- *Unified schemas:* Information becomes distributed among many different databases. One virtual repository defines (passive) consistent conversion mappings to multiple databases.

- *Multi-database:* Databases have many users; passive mappings don't suffice; active agents mediate interactions. *Layered hierarchy* with client-server interaction.

In this evolution, technological progress and expanding demand drive progress. Larger memories and faster processing enable access to an ever-wider assortment of data resources in a heterogeneous, distributed world. Our ability to exploit this remains limited by volume, complexity of mappings, the need to handle data discrepancies, and the need for sophisticated interpretation of requests for services and of available data.

3. Integration in Software Development Environments

Software development has relied on software tools for almost as long as data processing has relied on on-line databases. Initially these tools only supported the translation from source code to object code; they included compilers, linkers, and libraries. As time passed, many steps in the software development process became sufficiently routine to be partially or wholly automated, and tools now support analysis, configuration control, debugging, testing, and documentation as well. As with databases, the individual tools grew up independently. Although the integration problem has been recognized for nearly two decades (Toronto 1974), individual tools still work well together only in isolated cases.

3.1. Batch Sequential

The earliest software development tools were standalone programs. Often their output appeared only on paper and perhaps in the form of object code on cards or paper tape. Eventually most of the tools' results were at least in some magnetic—universally readable—form, but the output of each tool was most likely in the wrong format, the wrong units, or the wrong conceptual model for other tools to use. Even today, execution profiles are customarily provided in human-readable form but not propagated back to the compiler for optimization. Effective sharing of information was thus limited by lack of knowledge about how information was encoded in representations. As a result, manual translation of one tool's output to another tool's input format was common.

As time passed, new tools incorporated prior knowledge of related tools, and the usefulness of shared information became more evident. Scripts grew up to invoke tools in fixed orders. These scripts essentially defined batch sequential architectures.

This remains the most common style of integration for most environments. For example, in unix both shell scripts and make follow this paradigm. ASCII text is the universal exchange representation, but the conventions for encoding internal structure in ASCII remain idiosyncratic.

3.2. Transition from Batch Sequential to Repository

Our view of the architecture of a system can change in response to improvements in technology. The way we think about compilers illustrates this. In the 1970s, compilation was regarded as a sequential process, and the organization of a compiler was typically drawn as in Figure 14. Text enters at the left end and is transformed in a variety of ways—to lexical token stream, parse tree, intermediate code—before emerging as machine code on the right. We often refer to this compilation model as a pipeline, even though it was (at least originally) closer to a batch sequential architecture in which each transformation ("pass") ran to completion before the next one started.

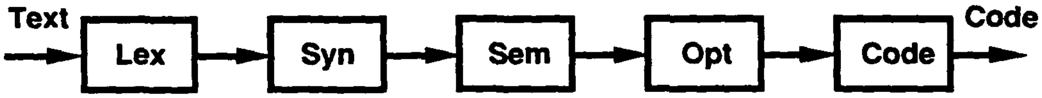


Figure 14: Traditional compiler model

In fact, even the batch sequential version of this model was not completely accurate. Most compilers created a separate symbol table during lexical analysis and used or updated it during subsequent passes. It was not part of the data that flowed from one pass to another but rather existed outside all the passes. So the system structure was more properly drawn as in Figure 15.

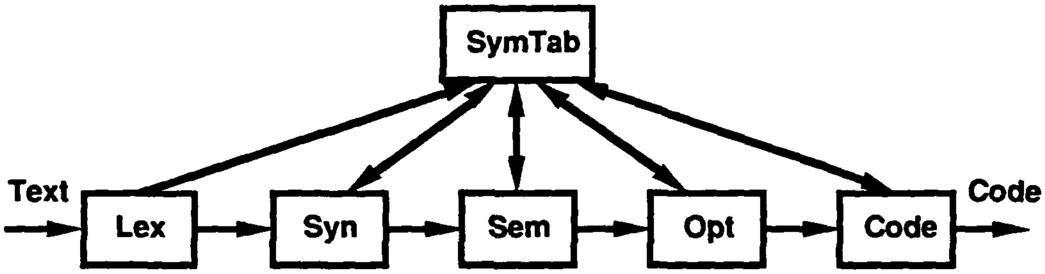


Figure 15: Traditional compiler model with symbol table

As time passed, compiler technology grew more sophisticated. The algorithms and representations of compilation grew more complex, and increasing attention turned to the intermediate representation of the program during compilation. Improved theoretical understanding, such as attribute grammars, accelerated this trend. The consequence was that by the mid-1980s the intermediate representation (for example, an attributed parse tree), was the center of attention. It was created early during compilation, manipulated during the remainder, and discarded at the end. The data structure might change in detail, but it remained substantially one growing structure throughout. However, we continued (sometimes to the present) to model the compiler with sequential data flow as in Figure 16.

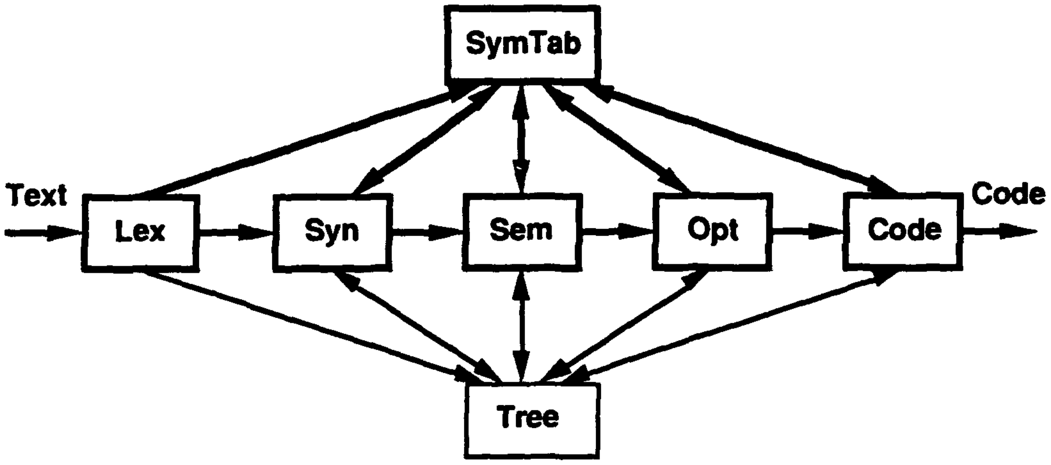


Figure 16: Modern canonical compiler

In fact, a more appropriate view of this structure would re-direct attention from the sequence of passes to the central shared representation. When you declare that the tree is the locus of compilation information and the passes define operations on the tree, it becomes natural to re-draw the architecture as in Figure 17. Now the connections between passes denote control flow, which is a more accurate depiction; the rather stronger connections between the passes and the tree/symbol table structure denote data access and manipulation. In this fashion, the architecture has become a repository, and this is indeed a more appropriate way to think about a compiler of this class.

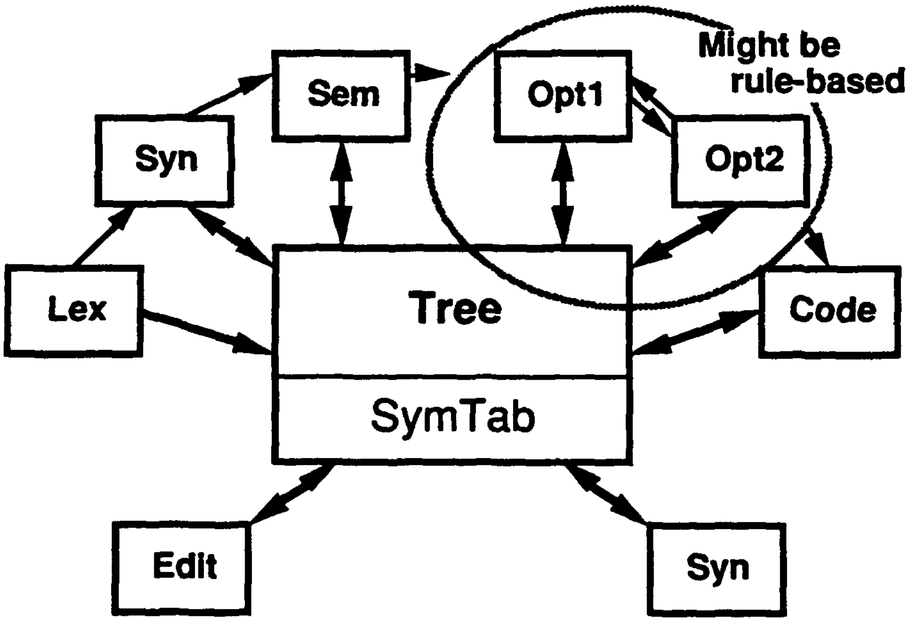


Figure 17: Repository view of modem compiler

Happily, this new view also accommodates various tools that operate on the internal representation rather than the textual form of a program; these include syntax-directed editors and various analysis tools.

Note that this repository resembles the database repository in some respects and differs in others. Like the database, the information of the compilation is localized in a central data component and operated on by a number of independent computations that interact only through the shared data. However, whereas the execution order of the operations in the database was determined by the types of the incoming transactions, the execution order of the compiler is predetermined, except possibly for opportunistic optimization.

3.3. Repository

Batch sequential tools and compilers—even when organized as repositories—do not retain information from one use to another. As a result, a body of knowledge about the program is not accumulated. The need for auxiliary information about a program to supplement the various source, intermediate, and object versions became apparent, and tools started retaining information about the prior history of a program.

The repository of the compiler provided a focus for this data collection. Efficiency considerations led to incremental compilers that updated the previous version of the augmented parse tree, and some tools came to use this shared representation as well. Figure 18 shows some of the ways that tools could interact with a shared repository.

- *Tight coupling:* Share detailed knowledge of the common, but proprietary, representation among the tools of a single vendor
- *Open representation:* Publish the representation so that tools can be developed by many sources. Often these tools can manipulate the data, but they are in a poor position to change the representation for their own needs.
- *Conversion boxes*: Provide filters that import or export the data in foreign representations. The tools usually lose the benefits of incremental use of the repository.
- *No contact:* Prevent a tool from using the repository, either explicitly, through excess complexity, or through frequent changes.

These alternatives have different functional, efficiency, and market implications.

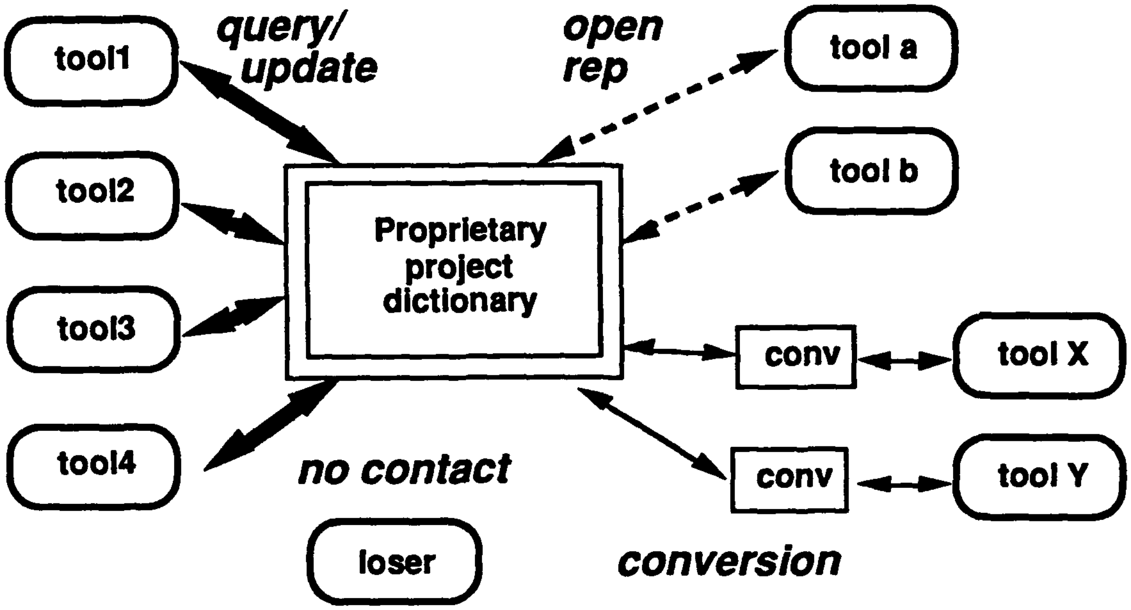
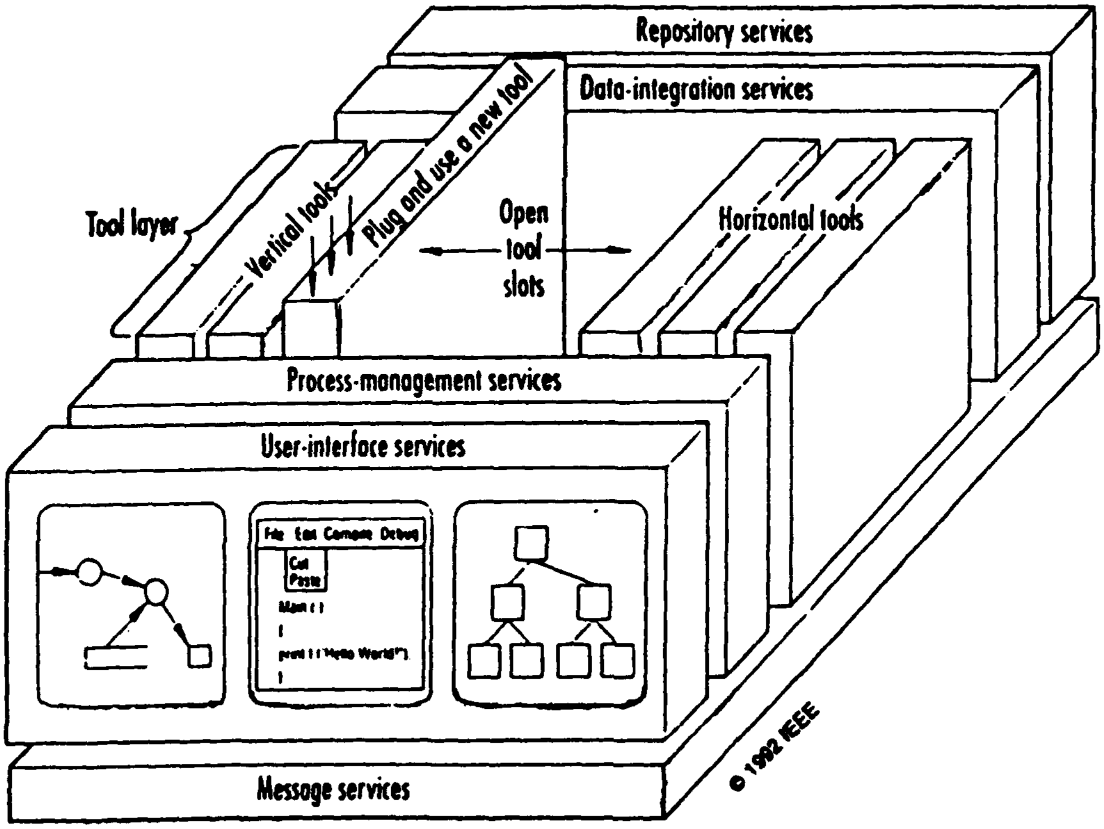


Figure 18: Software tools with shared representation

3.4. Hierarchical Layers

Current work on integration emphasizes interoperability of tools, especially in distributed systems. Figure 19 (Chen and Norman, 1992, p.19) shows one approach, the NIST/ECMA reference model. It resembles in some ways the layered architecture with mediators for databases, but it is more elaborate because it attempts to integrate communications and user interfaces as well as representation. It also embeds knowledge of software development processes, such as the order in which tools must be used and what situations call for certain responses.



© 1992 IEEE

Figure 19: NIST/ECMA reference model for environment integration

Note, however, that whereas this model provides for integration of data, it provides communication and user interface services directly. That is, this model allows for integration of multiple representations but fixes the models for user interfaces and communication.

In one variation on the integrated-environment theme, the integration system defined a set of "events" (e.g., "module foo.c recompiled") and provides support for tools to announce or to receive notice of the occurrence of events. This provides a means of communicating the need for action, but it does not solve the central problem of sharing information.

3.5. Evolution of Shared Information Systems in Software Development Environments

Software development has different requirements from database processing. As compared to databases, software development involves more different types of data, fewer instances of each distinct type, and slower query rates. The units of information are larger, more complex, and less discrete than in traditional databases. The lifetime of software development information, however, is not (or at least should not be) shorter than database lifetimes.

Despite the differences in application area and characteristics of the supporting data, the essential problem of collecting, storing, and retrieving shared data about an ongoing process is common to the two areas. It is therefore not surprising to find comparable evolutionary stages in their architectures.

Here the forces for evolution were

- the advent of on-line computing, which drove the shift from batch to interactive processing for many functions
- the concern for efficiency, which is driving a reduction in the granularity of operations, shifting from complete processing of systems to processing of modules to incremental development
- the need for management control over the entire software development process, which is driving coverage to increase from compilation to the full life cycle

Integration in this area is still incomplete. Data conversions are passive, and the ordering of operations remains relatively rigid. The integration systems can exploit only relatively coarse system information, such as file and date. Software development environments are under pressure to add capabilities for handling complex dependencies and selecting which tools to use. Steps toward more sophistication show up in the incorporation of metamodels to describe sharing, distribution, data merging, and security policies. The process-management services of the NIST/ECMA model are not yet well developed, and they will initially concentrate on project-level support. But integration across all kinds of information and throughout the life cycle is on the agenda, and intelligent assistance is often mentioned on the wish-list.

4. Integration in Building Design

The previous two examples come from the information technology fields. For the third example we turn to an application area, the building construction industry. This industry requires a diverse variety of expertise. Distinct responsibilities correspond to matching sets of specialized functions. Indeed, distinct subindustries support these specialties. A project generally involves a number of independent, geographically dispersed companies. The diversity of expertise and dispersion of the industry inhibit communication and limit the scope of responsibilities. Each new project creates a new coalition, so there is little accumulated shared experience and no special advantage for pairwise compatibility between companies. However, the subtasks interact in complex, sometimes non-obvious ways, and coordination among specialties (global process expertise) is itself a specialty (Terk 1992).

The construction community operates on divide-and-conquer problem solving with interactions among the subproblems. This is naturally a distributed approach; teams independent subcontractors map naturally to distributed problem-solving systems with coarse-grained cooperation among specialized agents. However, the separation into subproblems is forced by the need for specialization and the nature of the industry; the problems are not inherently decomposable, and the subproblems are often interdependent.

In this setting it was natural for computing to evolve bottom-up. Building designers have exploited computing for many years for tasks ranging from accounting to computer-aided design. We are concerned here with the software that performs analysis for various stages of the design activity. The 1960s and 1970s saw a number of algorithmic systems directed at aiding in the performance of individual phases of the facility development. However, a large number of tasks in facility development depend on judgment, experience, and rules of thumb accumulated by experts in the domain. Such tasks cannot be performed efficiently in an algorithmic manner (Terk 1992).

The early stages of development, involving standalone programs and batch-sequential compositions, are sufficiently similar to the two previous examples that it is not illuminating to review them. The first steps toward integration focused on support-supervisory systems, which provided basic services such as data management and information flow control to individual independent applications, much as software development environments did. The story picks up from the point of these early integration efforts.

Integrated environments for building design are frameworks for controlling a collection of standalone applications that solve part of the building design problem (Terk 1992). They must be

- efficient in managing problem-solving and information exchange
- flexible in dealing with changes to tools
- graceful in reacting to changes in information and problem solving strategies

These requirements derive from the lack of standardized problem-solving procedures; they reflect the separation into specialties and the geographical distribution of the facility development process.

4.1. Repository

Selection of tools and composition of individual results requires judgment, experience, and rules of thumb. Because of coupling between subproblems it is not algorithmic, so integrated systems require a planning function. The goal of an integrated environment is integration of data, design decisions, and knowledge. Two approaches emerged: the closely-coupled Master Builder, or monolithic system, and the design environment with cooperating tools. These early efforts at integration added elementary data management and information flow control to a tool-set.

The common responsibilities of a system for distributed problem-solving are:

- Problem partitioning (divide into tasks for individual agents)
- Task distribution (assign tasks to agents for best performance)
- Agent control (strategy that assures tasks are performed in organized fashion)
- Agent communication (exchange of information essential when subtasks interact or conflict)

The construction community operates on divide-and-conquer problem solving with interactions among the subproblems. This is naturally a distributed approach; teams independent subcontractors map naturally to distributed problem-solving systems with coarse-grained cooperation among specialized agents. However, the nature of the industry—its need for specialization—forces the separation into subproblems; the problems are not inherently decomposable, and the subproblems are often interdependent. This raises the control component to a position of special significance.

Terk (1992) surveyed and classified many of the integrated building design environments that were developed in the 1980s. Here's what he found:

- *Data:* mostly repositories: shared common representation with conversions to private representations of the tools
- *Communication:* mostly shared data, some messaging
- *Tools:* split between closed (tools specifically built for this system) and open (external tools can be integrated)
- *Control:* mostly single-level hierarchy; tools at bottom; coordination at top
- *Planning:* mostly fixed partitioning of kind and processing order; scripts sometimes permit limited flexibility

So the typical system was a repository with a sophisticated control and planning component. A fairly typical such system, IBDE (Fenves et al 1990) appears in Figure 20. Although the depiction is not typical, the distinguished position of the global data shows clearly the repository character. The tools that populate this IBDE are

- ARCHPLAN develops architectural plan from site, budget, geometric constraints
- CORE lays out building service core (elevators, stairs, etc.)
- STRYPES configures the structural system (e.g., suspension, rigid frame, etc.)
- STANLAY performs preliminary structural design and approximate analysis of the structural system.
- SPEX performs preliminary design of structural components.
- FOOTER designs the foundation.
- CONSTRUCTION PLANEX generates construction schedule and estimates cost.

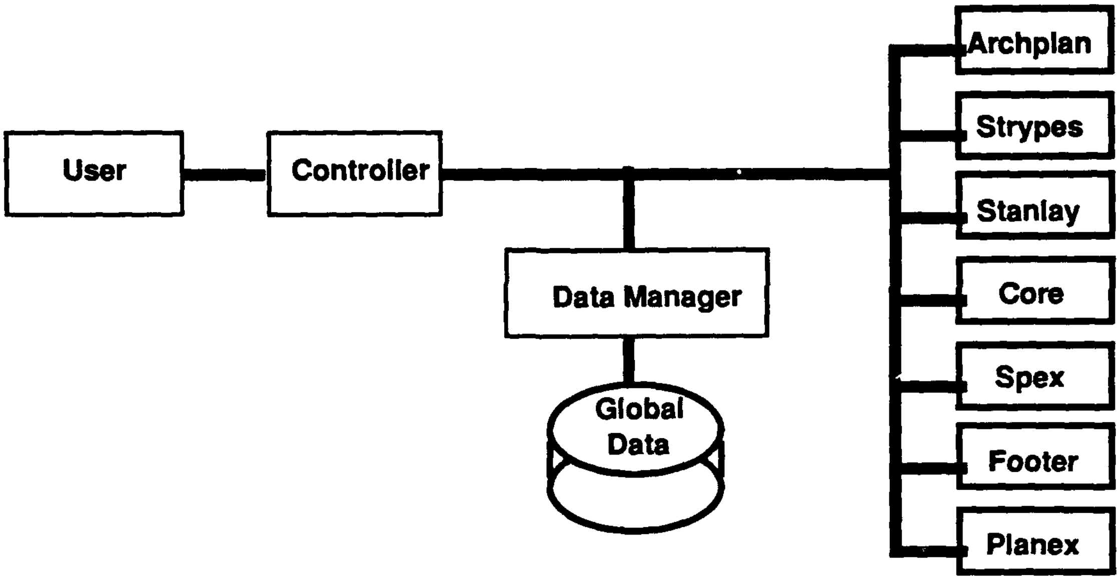


Figure 20: Integrated building design environment

4.2. Intelligent Control

As integration and automation proceed, the complexity of planning and control grows to be a significant problem. Indeed, as this component grows more complex, its structure starts to dominate the repository structure of the data. The difficulty of reducing the planning to pure algorithmic form makes this application a candidate for intelligent control.

The Engineering Design Research Center at CMU is exploring the development of intelligent agents that can learn to control external software systems, or systems intended for use with interactive human intervention. Integrated building design is one of the areas they have explored. Figure 22 (Newell and Steier 1991) shows their design for an intelligent extension of the original IBDE system, Soar/IBDE. That figure is easier to understand in two stages, so Figure 21 shows the relation of the intelligent agent to the external software systems before Figure 22 adds the internal structure of the intelligent agent. Figure 21 is clearly derived from Figure 20, with the global data moved to the status of just another external software system. The emphasis in Soar/IBDE was control of the interaction with the individual agents of IBDE.

From the standpoint of the designer's general position on intelligent control this organization seems reasonable, as the agent is portrayed as interacting with whatever software is provided. However, the global data plays a special role in this system. Each of the seven other components must interact with the global data (or else it makes no sense to retain the global data). Also, the intelligent agent may also find that the character of interaction with the global data is special, since it was designed to serve as a repository, not to interact with humans. Future enhancements of this system will probably need to address the interactions among components as well as the components themselves.

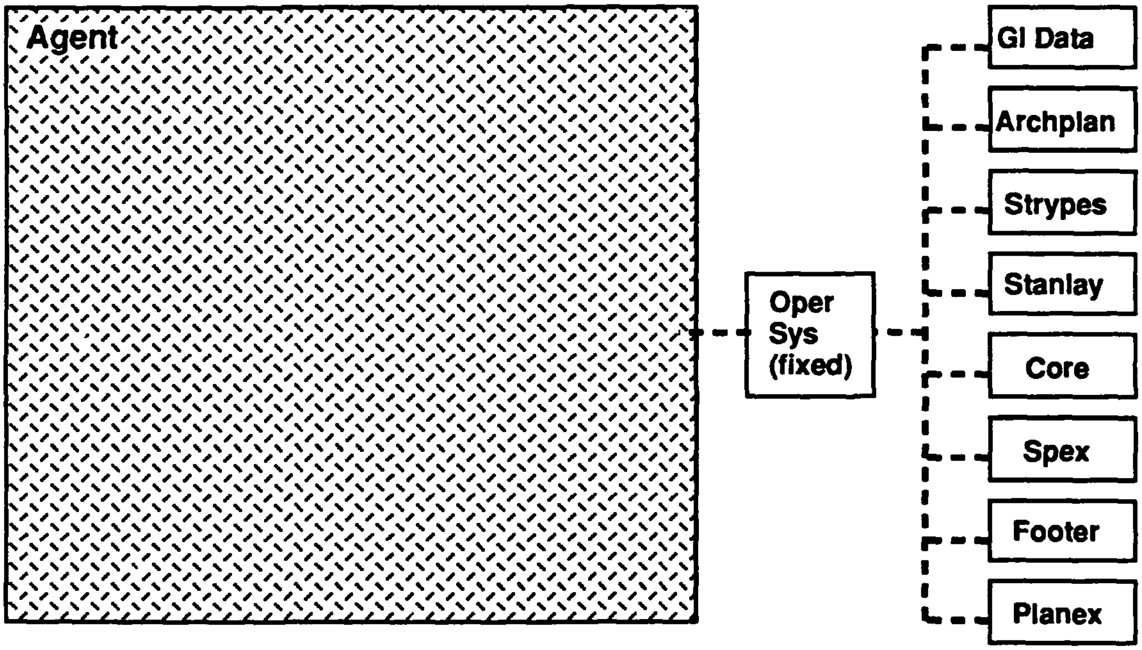


Figure 21: High-level architecture for intelligent IBDE

Figure 22 adds the fine structure of the intelligent agent. The agent has six major componerts It must be able to identify and formulate subtasks for the set of external software systems and express them in the input formats of those systems. It must receive the output and interpret it in terms of a global overview of the problem. It must be able to understand the actions of the components as they work toward solution of the problem, both in terms of general knowledge of the task and specific knowledge of the capabilities of the set of external software systems.

The most significant aspect of this design is that the seven external software systems are interactive. This means that their input and output are incremental, so a component that needs to understand their operation must retain and update a history of the interaction. The task becomes vastly more complex when pointer input and graphical output are included, though this is not the case in this case.

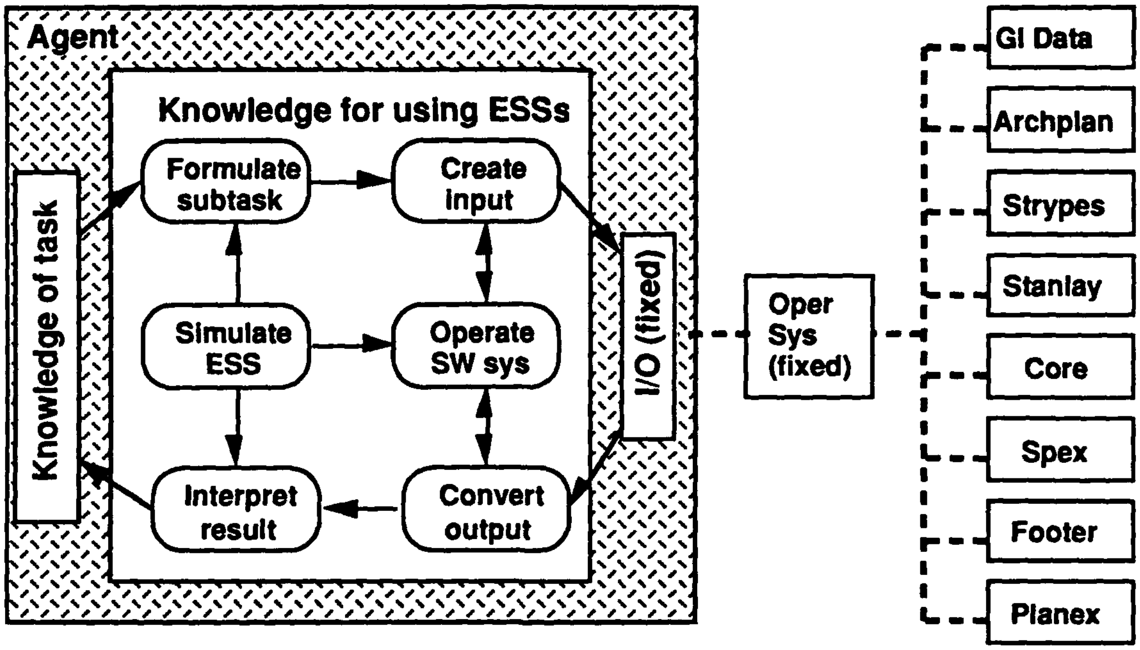


Figure 22: Detailed architecture for Soar/IBDE

4.3. Evolution of Shared Information Systems in Building Design

Integration in this area is less mature than in databases and software development environments. Nevertheless, the early stages of integrated building or facility environments resemble the early stages of the first two examples. The evolutionary shift to layered hierarchies seems to come when many users must select from a diverse set of tools and they need extra system structure to coordinate the effort of selecting and managing a useful subset. These systems have not reached this stage of development yet, so we don't yet have information on how that will emerge.

In this case, however, the complexity of the task makes it a prime candidate for intelligent control. This opens the question of whether intelligent control could be of assistance in the other two examples, and if so what form it will take. The single-agent model developed for Soar/IBDE is one possibility, but the enrichment of database mediators to make them able of independent intelligent action (like knowbots) is clearly another.

5. Architectural Structures for Shared Information Systems

While examining examples of software integration, we have seen a variety of general architectural patterns, or idioms for software systems. In this section we re-examine the data flow and repository idioms to see the variety that can occur within a single idiom.

Current software tools do not distinguish among different kinds of components at this level. These tools treat all modules equally, and they mostly assume that modules interact only via procedure calls and perhaps shared variables. By providing only a single model of component, they tend to blind designers to useful distinctions among modules. Moreover, by supporting only a fixed pair of low-level mechanisms for module interaction, they tend to blind designers to the rich classes of high-level interactions among components. These tools certainly provide little support for documenting design intentions in such a way that they become visible in the resulting software artifacts.

By making the richness of these structures explicit, we focus the attention of designers on the need for coherence and consistency of the system's design. Incorporating this information explicitly in a system design should provide a record that simplifies subsequent changes and increases the likelihood that later modifications will not compromise the integrity of the design. The architectural descriptions focus on design issues such as the gross structure of the system, the kinds of parts from which it is composed, and the kinds of interactions that take place.

The use of well-known patterns leads to a kind of reuse of design templates. These templates capture intuitions that are a common part of our folklore: it is now common practice to draw box-and-line diagrams that depict the architecture of a system, but no uniform meaning is yet associated with these diagrams. Many anecdotes suggest that simply providing some vocabulary to describe parts and patterns is a good first step.

By way of recapitulation, we now examine variations on two of the architectural forms that appear above: data flow and repositories.

5.1 Variants on Data Flow Systems

The data flow architecture that repeatedly occurs in the evolution of shared information systems is the batch sequential pattern. However, the most familiar example of this genre is probably the unix pipe-and-filter system. The similarity of these architectures is apparent in the diagrams used for systems of the respective classes, as indicated in Figure 23. Both decompose a task into a (fixed) sequence of computations. They interact only through the data passed from one to another and share no other information. They assume that the components read and write the data as a whole—that is, the input or output contains one complete instance of the result in some standard order. There are differences, though. Batch sequential systems are

- very coarse-grained
- unable to do feedback in anything resembling real time
- unable to exploit concurrency
- unlikely to proceed at an interactive pace

On the other hand, pipe-and-filter systems are

- fine-grained, beginning to compute as soon as they consume a few input tokens
- able to start producing output right away (processing is localized in the input stream)
- able to perform feedback (though most shells can't express it)
- often interactive

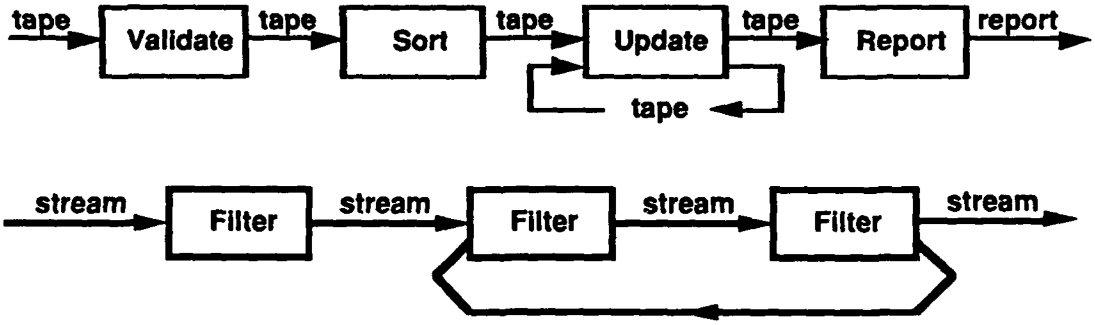


Figure 23 a, b: Comparison of (a) batch sequential and (b) pipe/filter architectures

5.2. Variants on Repositories

The other architectural pattern that figured prominently in our examples was the repository. Repositories in general are characterized by a central shared data store coupled tightly to a number of independent computations, each with its own expertise. The independent computations interact only through the shared data, and they do not retain any significant amount of private state. The variations differ chiefly in the control apparatus that controls the order in which the computations are invoked, in the access mechanisms that allow the computations access to the data, and in the granularity of the operations.

Figures 7 and 8 show a database system. Here the control is driven by the types of transactions in the input stream, the access mechanism is usually supported by a specialized programming language, and the granularity is that of a database transaction.

Figure 17 shows a programming language compiler. Here control is fixed (compilation proceeds in the same order each time), the access mechanism may be full conversion of the shared data structure into an in-memory representation or direct access (when components are compiled into the same address space), and the granularity is that of a single pass of a compiler.

Figure 18 shows a repository that supports independent tools. Control may be determined by direct request of users, or it may in some cases be handled by an event mechanism also shared by the tools. A variety of access methods are available, and the granularity is that of the tool set.

One prominent repository has not appeared here; it is mentioned now for completeness—to extend the comparison of repositories. This is the blackboard architecture, most frequently used for signal-processing applications in artificial intelligence (Nii 1986) and depicted in Figure 24. Here the independent computations are various knowledge sources that can contribute to solving the problem—for example, syntactic-semantic connection, phoneme recognition, word candidate generation, and signal segmentation for speech understanding. The blackboard is a highly-structured representation especially designed for the representations pertinent to the application. Control is completely opportunistic, driven by the current state of the data on the blackboard. The abstract model for access is direct visibility, as of many human experts watching each other solve a problem at a real blackboard (understandably, implementations support this abstraction with more feasible mechanisms). The granularity is quite fine, at the level of interpreting a signal segment as a phoneme.

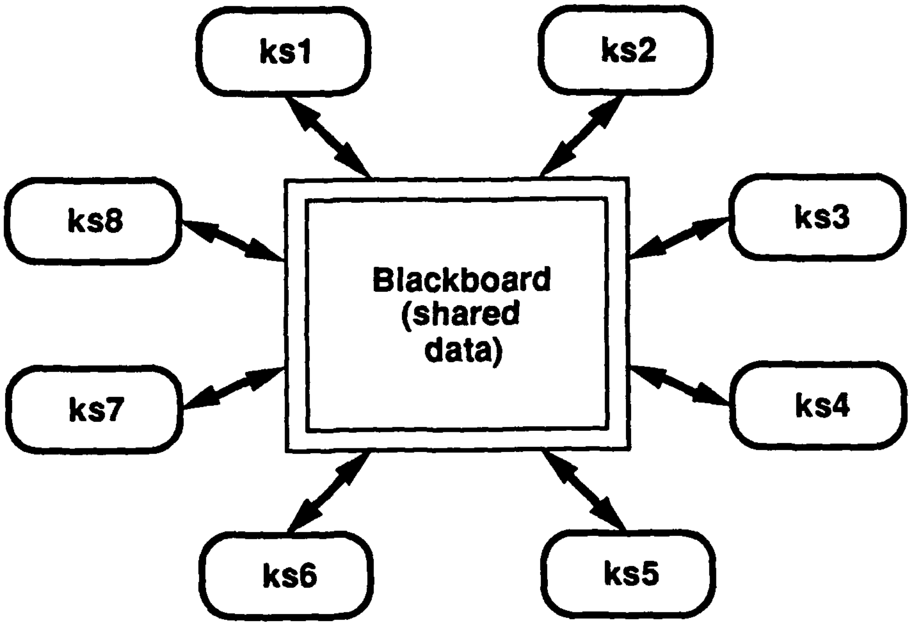


Figure 24: Blackboard architecture

6. Conclusions

Three tasks arising in different communities deal with collecting, manipulating, and preserving shared information. In each case changing technologies and requirements drove changes in the architectural form commonly used for the systems. We can identify that sequence as a common evolutionary pattern for shared information systems:

- isolated applications without interaction
- batch sequential processing
- repositories for integration via shared data
- layered hierarchies for dynamic integration across distributed systems

Since problems remain and new technology continues to emerge, this pattern may grow in the future, for example to add active control by intelligent agents.

These examples show one case in which a common problem structure appears in several quite different application areas. This suggests that attempts to exploit "domain knowledge" in software design should characterize domains by their computational requirements—e.g., shared information systems—as well as by industry—e.g., data processing, software development, or facility design. In addition, the examples show that within a single domain, differences among requirements or operational settings may change the preferred architecture. Taken together, this suggests that the notion of a single domain-specific architecture serving a segment of an industry may not fully exploit our growing architectural capabilities.

The models, notations, and tools for specifying software architectures remain informal. Although even informal models are useful, research in several areas is required to make these more precise and robust.

- Complete a taxonomy of common architectural patterns.
- Define and implement better abstractions for the interactions among components; at present system descriptions are cast in terms of procedure calls no matter what the abstractions may be.
- Establish ways to encapsulate stand-alone systems and express the resulting interfaces so they can be used as subsystems; linguistically this is a closure problem.
- Continue the exploration of independent agents for integration, especially in dynamically changing distributed systems.

Acknowledgments

This paper was written in honor of Allen Newell. Although we never collaborated directly, our professional interests touched from time to time, and these encounters often shaped the subsequent path of my research. This paper was so shaped at both ends: we discussed design levels many years ago, and I became acquainted with Soar/IBDE only a year or so ago. I also owe thanks to *David Steier for discussions about Soar/IBDE and to David Garlan for ongoing* collaboration in software architectures.

Several figures are reprinted by permission: Figure 1 from (Bell and Newell 1971), Gordon Bell. Figures 2, 3, 5, and 6 from (Best 1990) copyright © 1990 John Wiley and Sons Inc. Figures 9 from (Kim and Seo 1991), 10 from (Ahmed et al 1991), 12 from (Wiederhold 1992), and 19 from (Chen and Norman 1992) copyright © 1991, 1992 IEEE. Figures 20 and Figure 22 from (Newell and Steier), David Steier.

References

(Ahmed et al 1991) Rafi Ahmed et al. "The Pegasus Heterogeneous Multidatabase System." *IEEE Computer*, December 1991, 24, 12, pp. 19-27.

(Bell and Newell 1971) C. Gordon Bell and Allen Newell. *Computer Structures: Readings and Examples.* McGraw-Hill 1971.

(Best 1990) Laurence J. Best. *Application Architecture: Modern Large-Scale Information Processing.* Wiley 1990.

(Chen and Norman 1992) Minder Chen and Ronald J. Norman. "A Framework for Integrated CASE." *IEEE Software*, March 1992, 9, 2, pp. 18-22.

(CSTB 1992) Computer Science and Telecommunications Board. *Keeping the US Computer Industry Competitive: Systems Integration.* National Academy Press 1992.

(Fenves et al 1990) S. J. Fenves et al. "An Integrated Software Environment for Building Design and Construction." *Computer-Aided Design*, 22, 1, pp. 27-36.

(Garlan and Shaw 1993) David Garlan and Mary Shaw. "An Introduction to Software Architecture." In V. Ambriola and G. Tortora (eds.), *Advances in Software Engineering and Knowledge Engineering*, I, World Scientific Publishing Company, 1993 (to appear).

(Hovaness 1992) Haig Hovaness. "Price War: There's Fierce Combat Ahead over the Cost of Client-Server Databases." *Corporate Computing*, December 1992, 1, 6, pp. 45-46.

(Kim and Seo 1991) Won Kim and Jungyun Seo. "Classifying Schematic and Data Heterogeneity in Multidatabase Systems." *IEEE Computer*, December 1991, 24, 12, pp. 12-18.

(Newell 1982) Allen Newell. "The Knowledge Level." *Artificial Intelligence*, 1982, 18, pp. 87-127.

(Newell 1990) Allen Newell. *Unified Theories of Cognition.* Harvard University Press 1990.

(Newell and Steier 1991) Allen Newell and David Steier. "Intelligent Control of External Software Systems." *AI in Engineering*, to appear (was Carnegie Mellon University, Engineering Design Research Center Report EDRC 05-55-91).

(Nii 1986) H. Penny Nii. "Blackboard Systems." *AI Magazine*, 1986, 7, 3, pp. 38-53 and 7, 4, pp. 82-107.

(Nilsson et al 1990) Erik G. Nilsson et al. "Aspects of Systems Integration." *Systems Integration '90, Proc. First International Conference on Systems Integration*, 1990, pp. 434-413.

(Terk 1992) Michael Terk. A Problem-Centered Approach to Creating Design Environments for Facility Development. PhD Thesis, Civil Engineering Department, Carnegie Mellon University, 1992.

(Toronto 1974) *Proceedings of Workshop on the Attainment of Reliable Software.* University of Toronto, Toronto, Canada, June 1974.

(Wiederhold 1992) Gio Wiederhold. "Mediators in the Architecture of Future Information Systems." *IEEE Computer*, March 1992, 25, 3, pp. 38-48.

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-93-TR-3

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESC-TR-93-180

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (city, state, and zip code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (city, state, and zip code)
HQ ESC/ENS
Hanscom Air Force Base, MA 01731-2116

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESC/ENS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962890C0003

8c. ADDRESS (city, state, and zip code))
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT NO.
63756E	N/A	N/A	N/A

11. TITLE (Include Security Classification)
Software Architecture for Shared Information Systems

12. PERSONAL AUTHOR(S)
Mary Shaw

13a. TYPE OF REPORT
Final

13b. TIME COVERED
FROM TO

14. DATE OF REPORT (year, month, day)
March 1993

15. PAGE COUNT
46 pp.

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
Allen Newell
software architectures
info. systems

19. ABSTRACT (continue on reverse if necessary and identify by block number)

Software system design takes place at many levels. Different kinds of design elements, notations, and analyses distinguish these levels. At the *software architecture* level, designers combine subsystems into complete systems. This paper studies some of the common patterns of idioms, that guide these configurations. Results from software architecture offer some insight into the problems of systems integration–the task of connecting individual, isolated, pre-existing software systems to provide coherent, distributed solutions to large problems. As computing has become more sophisticated, so too have the software structures used in the integration task. This paper reviews historical examples of shared information systems in three different applications whose requirements share some common features about collecting, manipulating, and preserving large bodies of complex information. These applications have similar architectural histories in which a succession of designs responds to new technologies and new requirements for flexible, highly dynamic responses. A common pattern, the *shared information systems evolution pattern*, appears in all three

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT □ DTIC USERS ☒

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL
Thomas R. Miller, Lt Col, USAF

22b. TELEPHONE NUMBER (include area code)
(412) 268-7631

22c. OFFICE SYMBOL
ESC/ENS (SEI)

ABSTRACT — continued from page one, block 19

areas.

Technical Report
CMU/SEI-90-TR-22
ESD-90-TR-223

A Design Space and Design Rules for User Interface Software Architecture

Thomas G. Lane

November 1990

Technical Report

CMU/SEI-90-TR-22
ESD-90-TR-223
November 1990

A Design Space and Design Rules for User Interface Software Architecture

Thomas G. Lane

School of Computer Science
Software Architecture Design Principles Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF, SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This document is available through Research Access, Inc. / 800 Vinial Street / Pittsburgh, PA 15212. Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page at http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service / U.S. Department of Commerce / Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: 1-800-225-3842 or 703-767-8222.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

A Design Space and Design Rules for User Interface Software Architecture

Abstract. The architecture of a user interface software system can be described in terms of a fairly small number of key functional and structural choices. This report presents a "design space" that identifies these key choices and classifies the alternatives available for each choice. The design space is a useful framework for organizing and applying design knowledge. The report presents a set of design rules expressed in the terms of the design space. These rules can help a software designer to make good structural choices based on the functional requirements for a user interface system. Extension of this work might eventually provide automated assistance for structural design.

1. Introduction

Software architecture is the study of the large-scale structure and performance of software systems [Shaw 89]. Important aspects of a system's architecture include the division of functions among system modules, the means of communication between modules, and the representation of shared information. This report describes the architecture of user interface systems, using a *design space* that identifies the key architectural choices and classifies the available alternatives. The space provides a framework for *design rules* that can assist a designer in choosing an architecture that is appropriate for the functional requirements of a new system. The design space is useful in its own right as a shared vocabulary for describing and understanding systems.

This report is a summary of results from the author's thesis [Lane 90a]. It concentrates on presenting those results that are of interest to user interface system builders. A companion report argues that design spaces and rules may be a widely applicable means of expressing software engineering knowledge [Lane 90b].

1.1. Rationale

The established fields of engineering have long distinguished between routine and innovative design methods. *Routine design* uses standardized methods to solve problems similar to those that have been solved before. This process is not expected to yield the best possible design, but rather to yield a design that meets all the stated requirements with minimum design effort. In contrast, *innovative design* methods rely less on prior practice than on raw invention or derivation from abstract principles. Innovative designs can solve new types of problems or produce solutions especially well-tuned to specific requirements, but at a high design cost. Moreover, innovative design is more likely to fail to produce a solution than routine design (where a routine method is applicable). Engineering handbooks (e.g., [Perry 84]) exist primarily to support routine design. A good handbook arms its user with a number of standard design approaches and with knowledge of their strengths and limitations.

Routine design methods have benefits beyond reducing initial design cost. A standardized, commonly known design method reduces the effort needed to understand another person's design; hence, maintenance costs are also reduced. More fundamentally, standardized methods provide a context for the creation and application of knowledge; this is why a standardized method is usually better understood and more reliable than an ad hoc one. For example, the recognition and use of standard control flow patterns (conditionals, iteration, and so forth) made it possible for researchers to discover the key properties of those patterns (e.g., invariant and termination conditions of loops). Programmers now routinely use this knowledge to produce better-quality code than was possible without it.

At present, routine design is not well practiced by software engineers. Some designers tend to invent every system from scratch, while others tend to reuse a familiar design regardless of its suitability. Both errors arise from lack of a set of standardized methods. Handbook-like texts are now widely available for selection of algorithms and data structures (e.g., [Knuth 73, Sedgewick 88]), but such handbooks do not yet exist for higher levels of software design. The work reported here is a start toward developing a routine practice of software system architecture, within the limited domain of user interface systems.

The systems covered by this study are those whose main focus is on providing an interactive user interface for some software function(s). This includes user interface management systems (UIMSs), graphics packages, user interface toolkits, window managers, and even stand-alone applications that have a large user interface component. This range is large enough that no single design can cover all cases; hence, we must consider how to choose among alternatives. At the same time, the range is not too large to allow recognition of common patterns. Future work may make it possible to construct useful design spaces for larger classes of software systems.

1.2. The Notion of a Design Space

The central concept in this report is that of a multi-dimensional design space that classifies system architectures. Each dimension of a design space describes variation in one system characteristic or design choice. Values along a dimension correspond to alternative requirements or design choices. For example, required response time could be a dimension; so could the means of interprocess synchronization (e.g., messages or semaphores). A specific system design corresponds to a point in the design space, identified by the dimensional values that correspond to its characteristics and structure. Figure 1-1 illustrates a tiny design space.

A design dimension is not necessarily a continuous scale; in most cases the space considers only a few discrete alternatives. For example, methods for specifying user interface behavior include state transition diagrams, context-free grammars, menu trees, and many others. Each of these techniques has many small variations, so one of the key problems in constructing a design space is finding the most useful granularity of classification. Even when a dimension is in principle continuous, one may choose to aggregate it into a few

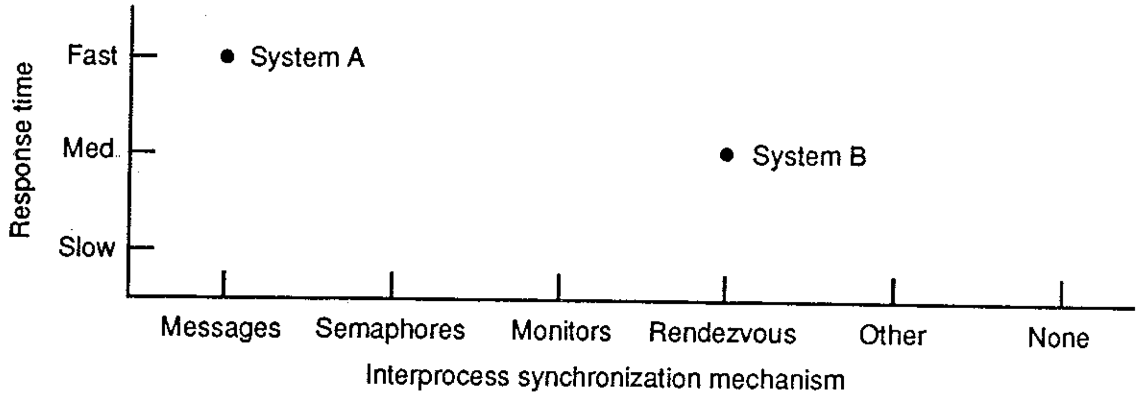


Figure 1-1: A Simple Design Space

discrete values (e.g., "low," "medium," "high"). This is appropriate when such gross estimates provide as much information as one needs (or can get) in the early stages of design.

Another way in which a design space differs from geometric intuition is that the dimensions may not be independent. In fact, it is important to discover correlations between the dimensions in order to create design rules describing appropriate and inappropriate combinations of choices. One empirical way of discovering such correlations is to see whether successful system designs cluster in some parts of the space and are absent from others.

A crucial part of this approach is to choose some dimensions that reflect requirements or evaluation criteria (function and/or performance), as well as other dimensions that reflect structure (or other available design choices). Then, the observed correlations and resulting design rules can provide direct design guidance: they show which design choices are most likely to meet the functional requirements for a new system.

1.3. Related Work

The concept of a design space is far from new. A seminal use is Bell and Newell's taxonomy of computer hardware structures [Bell 71]. They describe computers using dimensions such as function (e.g., numeric calculation or communication), instructions per second, memory size, and hardware-supported data types. A software-oriented example is Wegner's design space for object-oriented languages [Wegner 87].

The domain covered by this report's design space is user interface software. Various researchers have investigated individual aspects of user interface software structures. Most prior work deals with control flow patterns [Hayes 85, Tanner 83] or classification of notations for user interface appearance and behavior [Green 86, Myers 89]. Other workers have made proposals for standard module structures [Dance 87, Lantz 87]. Hartson and Hix survey much of the existing work [Hartson 89]. For the most part, however, the user interface research community has neglected internal structural issues in favor of work on selection and description of the external behavior of a user interface. Hence the work reported here provides a more complete view of the space of user interface structural alternatives than any prior work and, for several of the previously investigated dimensions, it offers new classifications that are more useful for making structural decisions.

2. An Overview of the Design Space

This section introduces the design space by describing a dozen of its dimensions. The complete space includes nearly fifty dimensions, many of which are fairly obvious to anyone familiar with user interface software. To avoid bogging down in details, we will consider only the more interesting dimensions. For a complete description of the space, see Appendix A.

2.1. A Basic Structural Model

To describe structural alternatives, it is necessary to have some terminology that identifies components of a system. The terminology must be quite general, or it will be inapplicable to some structures. A useful scheme for user interface systems divides any complete system into three components, or groups of modules:

1. An **application-specific** component: Code that is specific to one particular application program and is not intended to be reused in other applications. In particular, this component includes the functional core of the application. It may also include application-specific user interface code. (The term "code" should be read as including tables, grammars, and other non-procedural specifications, as well as conventional programming methods.)
2. A **shared user interface** component: Code that is intended to support the user interface of multiple application programs. If the software system can accommodate different types of I/O devices, only code that is applicable to all device types is included here.
3. A **device-dependent** component: Code that is specific to a particular I/O device class (and is not application-specific).

In a simple system, the second or third component might be empty: there might be no shared code other than device drivers, or the system might have no provision for supporting multiple device types (and hence no clear demarcation of device-specific code).

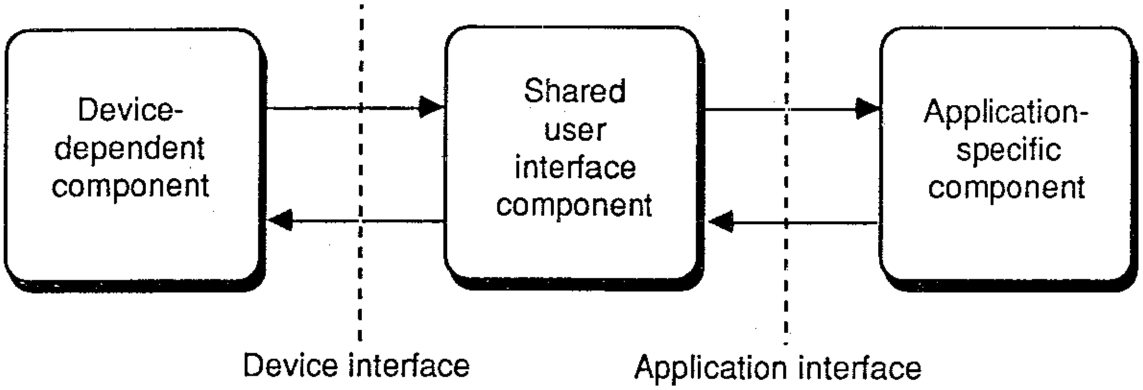

Device-dependent component
Shared user interface component
Application-specific component
Device interface
Application interface

Figure 2-1: A Basic Structural Model for User Interface Software

The intermodule divisions that the design space considers are the division between application-specific code and shared user interface code on the one hand, and between device-specific code and shared user interface code on the other. These divisions are called the *application interface* and *device interface* respectively. Figure 2-1 illustrates the structural model.

There is some flexibility in dividing a real system into these three components. This apparent ambiguity is very useful, for one can analyze different levels of the system by adopting different labelings. For example, in the X Window System [Scheifler 86], one may analyze the window server's design by regarding everything outside the server as application specific, then dividing the server into shared user interface and device-dependent levels. To analyze an X toolkit package, it is more useful to label the toolkit as the shared code, while regarding the server as a device-specific black box.

2.2. Functional Design Dimensions

The functional dimensions identify the user interface system requirements that most affect the system's structure. These dimensions are not intended to correspond to the earliest requirements that one might write for a system, but rather to identify the specifications that immediately precede the gross structural design phase. Thus, some design decisions have already been made in arriving at these choices.

The first example of a functional dimension is **command execution time**. This dimension indicates how long the application program may take to process a command, compared with the reaction time of a human user. Useful classifications are:

- **Short maximum time:** All commands can be executed in a short time, say a few tenths of a second.
- **Intermediate maximum, short average:** Most commands are executed in a short time, but some may take a bit longer, up to a couple of seconds.
- **Long maximum time:** Some or all commands may take a long time to execute so that the user will have a strong perception of waiting.

As an example of the importance of command execution time, a system in the first category can probably dispense with handling asynchronous input (i.e., no type-ahead or command cancellation features). This is less likely to be appropriate when long-running commands are present.

The second example functional dimension is **external event handling**: does the application program need to respond to external events, that is, events not originating in the user interface? If so, on what time scale?

- **No external events:** The application is uninfluenced by external events, or checks for them only as part of executing specific user commands. For example, a mail program might check for new mail, but only when an explicit command to do so is given. In this case, no support for external events is needed in the user interface.
- **Process events while waiting for input:** The application must handle exter-

nal events, but response time requirements are not so stringent that it must interrupt processing of user commands. It is sufficient for the user interface to allow response to external events while waiting for input.

- **External events preempt user commands:** External event servicing has sufficiently high priority that user command execution must be interrupted when an external event occurs.

Like the previous dimension, external event handling has obvious implications for control flow within the user interface and application.

The third example of a functional dimension is **user interface adaptability across devices**. This dimension measures how much change in user interface behavior may be required when changing to a different set of I/O devices:

- **None:** All aspects of behavior are the same across all supported devices. (This includes the case that only one set of I/O devices is supported.)
- **Local behavior changes:** Only changes in small details of behavior across devices; for example, the appearance of menus.
- **Global behavior changes:** Major changes in surface user interface behavior; for example, a change between menu-driven and command-language interface types.
- **Application semantics changes:** Changes in underlying semantics of commands (e.g., continuous display of state versus display on command).

The final examples are a complementary pair of dimensions. **Application portability across interaction styles** specifies the degree of portability across interaction styles required for applications that will use the user interface software:

- **High:** Applications should be portable across significantly different styles (e.g., command language versus menu-driven).
- **Medium:** Applications should be independent of minor stylistic variations (e.g., menu appearance).
- **Low:** User interface variability is not a concern, or application changes are acceptable when modifying the user interface.

User interface system adaptability across interaction styles specifies how adaptable to different interaction styles the shared user interface software should be:

- **High:** Adaptable to a wide range of interface styles.
- **Medium:** Limited adaptability.
- **Low:** Imposes a specific interface style.

Since interface behavior must be specified somewhere, there is a tradeoff between application and shared user interface flexibility: either the shared software imposes a stylistic decision, or the application makes the decision and hence becomes less portable. This dilemma can be alleviated by wise use of default choices, but in general, high requirements for both of these dimensions should be viewed with suspicion. In the other direction, low requirements for both dimensions indicate little flexibility in user interface behavior, which is perfectly appropriate for some systems (for example, if strong user interface conventions exist).

2.2.1. The Most Important Functional Dimensions

It is reasonable to expect that some functional dimensions have more influence on structure than others, but it is difficult to guess which ones have the greatest impact. Some insight can be gained from the author’s experiments with automated design rules (see Section 4): we can rank the functional dimensions according to the total weight given to each in the automated rule set. (Those rules did not fully reproduce the decisions of human experts, so this ranking may need to be modified when better data is available.) On this basis, the five functional dimensions with most influence on the structural dimensions are:

- User interface system adaptability across devices
- Application portability across devices
- Application portability across interaction styles
- Basic interface class
- System organization

The next five dimensions are:

- Available processing power
- I/O device class breadth
- User interface system adaptability across interaction styles
- User customizability
- External event handling

The remaining fifteen functional dimensions (listed in Appendix A) have less influence on structure.

The most striking feature of this ranking is the importance of dimensions having to do with flexibility. Evidently the nature and degree of adaptability required of the system are by far the most important determinants of an appropriate structure. It is an open question whether this property is unique to user interface system structures, or is true for other kinds of software as well.

2.3. Structural Design Dimensions

This section presents some important structural dimensions: the fundamental decisions about system structure.

Application interface abstraction level is in many ways the key structural dimension. The design space identifies six general classes of application interface, which are most easily distinguished by the level of abstraction in communication:[1]

- **Monolithic program:** There is no separation between application-specific and shared code, hence no application interface (and no device interface, either). This can be an appropriate solution in small, specialized systems where the application needs considerable control over user interface details and/or little processing power is available. (Video games are a typical example.)
- **Abstract device:** The shared code is simply a device driver, presenting an abstract device for manipulation by the application. The operations provided have specific physical interpretations (e.g., "draw line," but not "present menu"). Most aspects of interactive behavior are under the control of the application, although some local interactions may be handled by the shared code (e.g., character echoing and backspace handling in a keyboard/display driver). In this category, the application interface and device interface are the same.
- **Toolkit:** The shared code provides a library of interaction techniques (e.g., menu or scroll bar handlers). The application is responsible for selecting appropriate toolkit elements and composing them into a complete interface; hence the shared code can control only local aspects of user interface style, with global behavior remaining under application control. The interaction between application and shared code is in terms of specific interactive techniques (e.g., "obtain menu selection"). The application can bypass the toolkit, reaching down to an underlying abstract device level, if it requires an interaction technique not provided by the toolkit. In particular, conversions between specialized application data types and their device-oriented representations are done by the application, accessing the underlying abstract device directly.[2]
- **Interaction manager with fixed data types:** The shared code controls both local and global interaction sequences and stylistic decisions. Its interaction with the application is expressed in terms of abstract information transfers, such as "get command" or "present result" (notice that no particular external representation is implied). These abstract transfers use a fixed set of standard data types (e.g., integers, strings); the application must express its input and output in terms of the standard data types. Hence some aspects of the conversion between application internal data formats and user-visible representations remain in the application code.

[1] Recognition of abstraction level as a key property in user interfaces goes back at least to Hayes et al [Hayes 85]. The classification used here is a practical one, but it is based on the theoretical distinctions made by Hayes.

[2] The notion that conversion between internal and external representations of data types is a key activity in user interfaces is due to Shaw [Shaw 86].

- **Interaction manager with extensible data types:** As above, but the set of data types used for abstract communication can be extended. The application does so by specifying (in some notation) the input and output conversions required for the new data types. If properly used, this approach allows knowledge of the external representation to be separated from the main body of the application.
- **Extensible interaction manager:** Communication between the application and shared code is again in terms of abstract information transfers. The interaction manager provides extensive opportunities for application-specific customization. This is accomplished by supplying code that augments or overrides selected internal operations of the interaction manager. (Most existing systems of this class are coded in an object-oriented language, and the language's inheritance mechanism is used to control customization.) Usually a significant body of application-specific code customizes the interaction manager; this code is much more tightly coupled to the internal details of the interaction manager than is the case with clients of nonextensible interaction managers.

This classification turns out to be sufficient to predict most aspects of the application interface, including the division of user interface functions, the type and extent of application knowledge made available to the shared user interface code, and the kinds of data types used in communication. For instance, we have already suggested the division of local versus global control of interactive behavior that is typically found in each category.

Abstract device variability is the key dimension describing the device interface. We view the device interface as defining an *abstract device* for the device-independent code to manipulate. The design space classifies abstract devices according to the degree of variability perceived by the device-independent code:

- **Ideal device:** The provided operations and their results are well specified in terms of an "ideal" device; the real device is expected to approximate the ideal behavior fairly closely. (An example is the PostScript imaging model, which ignores the limited resolution of real printers and displays [Adobe 85].) In this approach, all questions of device variability are hidden from software above the device driver level, so application portability is high. This approach is most useful where the real devices deviate only slightly from the ideal model, or at least not in ways that require rethinking of user interface behavior.
- **Parameterized device:** A class of devices is covered, differing in specified parameters such as screen size, number of colors, number of mouse buttons, etc. The device-independent code can inquire about the parameter values for the particular device at hand, and adapt its behavior as necessary. Operations and their results are well specified, but depend on parameter values. (An example is the X Windows graphics model, which exposes display resolution and color handling [Scheifler 86].) The advantage of this approach is that higher level code has both more knowledge of acceptable tradeoffs and more flexibility in changing its behavior than is possible for a device driver. The drawback is that device-independent code may have to perform complex case analysis in order to handle the full range of supported devices. If this must be done in each application, the cost is high and there is a great risk that programmers will omit support for some devices. To reduce this temptation, it is best to design a

parameterized model to have just a few well-defined levels of capability, so as to reduce the number of cases to be considered.

- **Device with variable operations:** A well-defined set of device operations exists, but the device-dependent code has considerable leeway in choosing how to implement the operations; device-independent code is discouraged from being closely concerned with the exact external behavior. Results of operations are thus not well specified. (For example, GKS logical input devices [Rosenthal 82] and the Scribe formatting model [Reid 80].) This approach works best when the device operations are chosen at a level of abstraction high enough to give the device driver considerable freedom of choice. Hence the device-independent code must be willing to give up much control of user interface details. This restriction means that direct manipulation (with its heavy dependence on semantically-controlled feedback) is not well supported.
- **Ad-hoc device:** In many real systems, the abstract device definition has developed in an ad-hoc fashion, and so it is not tightly specified; behavior varies from device to device. Applications therefore must confine themselves to a rather small set of device semantics if they wish to achieve portability, even though any particular implementation of the abstract device may provide many additional features. (Alphanumeric terminals are an excellent example.) While aesthetically displeasing, this approach has one redeeming benefit: applications that do not care about portability are not hindered from exploiting the full capabilities of a particular real device.

These categories lend themselves to different situations. For example, abstract devices with variable operations are useful when much of the system’s "intelligence" is to be put into the device-specific layer; but they are only appropriate for handling local changes in user interface behavior across devices.

Notation for user interface definition classifies the techniques used for defining user interface appearance and behavior:

- **Implicit in shared user interface code:** Information "wired into" shared code. For example, the visual appearance of a menu might be implicit in the menu routines supplied by a toolkit. In systems where strong user interface conventions exist, this is a perfectly acceptable approach.
- **Implicit in application code:** Information buried in the application and not readily available to shared user interface code. This is most appropriate where the application is already tightly involved in the user interface, for example, in handling semantic feedback in direct manipulation systems.
- **External declarative notation:** A nonprocedural specification separate from the body of the application program, for example, a grammar or tabular specification. External declarative notations are particularly well suited to supporting user customization and to use by nonprogramming user interface experts. **Graphical specification** methods are an important special case.
- **External procedural notation:** A procedural specification separate from the body of the application program; often cast in a specialized programming language. Procedural notations are more flexible than declarative ones, but are harder to use. User-accessible procedural mechanisms, such as macro defini-

tion capability or the programming language of EMACS-like editors [Borenstein 88], provide very powerful customization possibilities for sophisticated users. However, an external notation by definition has limited access to the state of the application program, which may restrict its capability.

- **Internal declarative notation:** A nonprocedural specification within the application program. This differs from an implicit representation in that it is available for use by the shared user interface code. Parameters supplied to shared user interface routines often amount to an internal declarative notation. An example is a list of menu entries provided to a toolkit menu routine.
- **Internal procedural notation:** A procedural specification within the application program. This differs from an implicit representation in that it is available for use by the shared user interface code. A typical example is a status-inquiry or data transformation function that is provided for the user interface code to call. This is the most commonly used notation for customization of extensible interaction managers. It provides an efficient and flexible notation, but is not accessible to the end user, and so is useless for user customization. It is particularly useful for handling application-specific feedback in direct manipulation interfaces, since it has both adequate flexibility and efficient access to application semantics.

Each of these categories offers a different tradeoff between power, runtime cost, ease of use, and ease of modification. For example, declarative notations are the easiest to use (especially for nonprogramming user interface designers) but have the least power, since they can only represent a predetermined range of possibilities. Typically, several notational techniques are used in a system, with different aspects of the user interface controlled by different techniques. For example, the position and size of a screen button might be specified graphically, while its highlighting behavior is specified implicitly by the code of a toolkit routine.

Application control flow indicates where input processing occurs in the application's flow of control:

- **Single input point:** The system contains an *event loop* that is the sole point at which user input is accepted; when an input event is received, it is processed; then control returns to the event loop to await the next input. Note that the event loop may be in either application or shared code.
- **Multiple input point:** Input is accepted at multiple points in the application's flow of control. (Usually, each such point can handle only a subset of the possible inputs, leading to modal interface behavior.)

This classification is a variation of the standard distinction between "internal control" (application calls user interface) and "external control" (user interface calls application) [Hayes 85, Tanner 83]. The standard terminology is unsatisfactory because the properties usually associated with external control actually apply to any system using an event loop, regardless of the direction of subroutine calls.

Number of control threads indicates how many logical threads of control exist in the application and user interface:

- **Single thread of control.**
- **One user interface thread and one application thread.**
- **Multiple user interface threads and one application thread.**
- **One user interface thread and multiple application threads.**
- **Multiple user interface threads and multiple application threads.**

Multiple threads are useful for dealing with external events or logically independent concurrent dialogues (e.g., multiple input devices). The one-plus-one-thread choice is particularly simple and helpful for decoupling application processing (including external event handling) from user interface logic.

Control thread mechanism describes the method, if any, used to support multiple logical threads of control. Often, full-fledged processes are too difficult to implement or impose too much overhead, so many partial implementations are used. This dimension classifies the possibilities as follows:

- **None:** Only a single logical control thread is used.
- **Standard processes:** Independently scheduled entities with interprocess protection (typically, separate address spaces). These provide security against other processes, but interprocess communication is relatively expensive. For a user interface system, security may or may not be a concern, while communication costs are almost always a major concern. In network environments, standard processes are usually the only kind that can be executed on different machines.
- **Lightweight processes:** Independently scheduled entities within a shared address space. These are only suitable for mutually trusting processes due to lack of security; but often that is not a problem for user interface systems. The benefit is substantially reduced cost of communication, especially for use of shared variables. Few operating systems provide lightweight processes, and building one’s own lightweight process mechanism can be difficult.
- **Non-preemptive processes:** Processes without preemptive scheduling (must explicitly yield control), usually in a shared address space. These are relatively simple to implement. Guaranteeing short response time is difficult and impacts the entire system: long computations must be broken up explicitly.
- **Event handlers:** Pseudo-processes which are invoked via a series of subroutine calls; each such call must return before another event handler process can be executed. Hence control flow is restricted; in particular, waiting for another process cannot occur inside a subroutine called by an event handler. Again, response time constraints require system-wide attention. The main advantage of this method is that it requires virtually no support mechanism.
- **Interrupt service routines:** Hardware-level event handling; a series of interrupt service routine executions form a control thread, but one with restricted control flow and communication abilities. The control flow restrictions are comparable to event handlers; but unlike event handlers, preemptive scheduling is available.

Event handlers are easily implemented within a user interface system; non-preemptive processes are harder but can still be implemented without operating system support. The other mechanisms usually must be provided by the operating system. Some form of preemptive scheduling is often desirable to reduce timing dependencies between threads.

Basis of communication classifies systems according to whether communication between modules depends upon shared state, upon events, or both. An *event* is a transfer of information occurring at a discrete time, for example, via a procedure call or message. Communication through shared state variables is significantly different, because the recipient always has access to the current values and need not use information in the same order in which it is sent. The design space recognizes four categories:

- **Events:** There is no shared state; all communication relies on events.
- **Pure state:** Communication is strictly via shared state; the recipient must repeatedly inspect the state variables to detect changes.
- **State with hints:** Communication is via shared state, but the recipient is actively informed of changes via an event mechanism; hence polling of the state is not required. However, the recipient could ignore the events and reconstruct all necessary information from the shared state; so the events are efficiency hints rather than essential information.
- **State plus events:** Both shared state and events are used; the events are crucial because they provide information not available from state monitoring.

State-based mechanisms are popular for dealing with incrementally updated displays. The hybrid state/event categories provide possibilities for performance optimization in return for their extra complexity. State-based communication requires access to shared storage, which may be impossible or unreasonably expensive in some system architectures.

It is possible for different bases of communication to be used at the application and device interfaces, but this is rare. It is fairly common to have different bases of communication for input and output; hence the design space provides separate dimensions for input and output communication basis.

3. Design Rules for User Interface Systems

This section presents some design rules that relate the functional and structural dimensions of the design space. Again, we will consider only a few sample rules to illustrate the flavor of the approach. For a more thorough presentation of the rules, see Appendix B.

Both here and in Appendix B, we present the rules in an informal fashion. In this form, the rules are directly useful as guidelines or rules of thumb for a human designer. Section 4 describes how the rules can be made more formal and suitable for use in automated design.

- Stronger requirements for user interface adaptability across devices favor higher levels of application interface abstraction, so as to decouple the application from user interface details that may change across devices. If the requirement is for global behavior or application semantics changes, then parameterized abstract devices are also favored. Such changes generally have to be implemented in shared user interface code or application code, rather than in the device driver; so information about the device at hand cannot be hidden from the higher levels, as the other classes of abstract device try to do. However, a requirement for local behavior changes can favor abstract devices with variable operations, since this method can allow all of the required adaptation to be hidden within the device driver.
- High user customizability requirements favor external notations for user interface behavior. Implicit and internal notations are usually more difficult to access and more closely coupled to application logic than are external notations.
- A high requirement for application portability across user interface styles favors the higher levels of application interface abstraction. Less obviously, it favors event-based or pure state-based communication over the hybrid forms (state with hints or state plus events). A hybrid communication protocol is normally tuned to particular communication patterns, which may change when user interface style changes.
- If the maximum command execution time is short, a single thread of control is practical and is favored as the simplest solution. With longer commands, multiple threads are favored to permit user input processing to continue; this is necessary to support command cancellation, for example.
- If external events must be handled, it is often worthwhile to provide separate control thread(s) for this purpose. Separate threads serve to decouple event handling logic from user interface logic. When external event handling requires preemption of user commands, a preemptive control thread mechanism (standard processes, lightweight processes, or interrupt service routines) is strongly favored. Without such a mechanism, very severe constraints must be placed on all user interface and application processing in order to guarantee adequate response time.
- The most commonly useful control thread mechanisms are standard processes, lightweight processes, and event handlers; the others are appropriate only in special cases. For most user interface work, lightweight processes are very appropriate if available. Standard processes should be used when protection considerations warrant, and in network environments where it may be useful to

put the processes on separate machines. If these conditions do not apply, event handlers are the best choice when their response time limitations are acceptable; otherwise it is probably best to invest in building a lightweight process mechanism.

The preceding rules all relate functional to structural dimensions. Following is an example of the rules interconnecting structural dimensions.

- The choice of application interface abstraction level influences the choice of notation for user interface behavior. In monolithic programs and abstract-device application interfaces, implicit representation is usually sufficient. In toolkit systems, implicit and internal declarative notations are found (parameters to toolkit routines being of the latter class). Interaction managers of all types use external and/or internal declarative notations. Extensible interaction managers rely heavily on procedural notations, particularly internal procedural notation, since customization is often done by supplying procedures.

The reader may well have found these rules to be fairly obvious and a bit boring. This is an indication of the conceptual power of the design space: many useful rules are immediate consequences of the properties of the chosen dimensions. Though straightforward, these rules are sufficiently powerful to be a useful aid to design.

4. Automating the Design Rules

The design rules are presented in this report in an informal fashion suitable for use as guidelines by human software designers. It is also possible to express the rules in a more detailed, rigorous formulation. In such a form the rules could be used as the basis for an automatic design aid. The author has experimented with such automated rules, with promising results.

The rules were expressed in the form of numerical weights associated with particular combinations of values along different dimensions. For example, the combination of no external events and single thread of control received a positive weight, indicating that a single thread of control may be a good choice given that requirement; while the combination of preemptive external events and single thread of control received a negative weight. Given a set of functional requirements and a proposed structural design, the weights of the applicable rules can be combined to give a score for that design. A straightforward search algorithm was used to find the highest-scoring design for a given set of requirements.

These automated rules were tested by comparing their recommendations to the actual design choices of expert human software designers, as expressed in a set of test cases. A moderate to substantial degree of agreement was observed. This preliminary result suggests that this approach has considerable potential for creating practical design aids. More immediately, it gives some confidence that the design space described here captures useful knowledge about user interface software design.

Additional information about this experiment is given in the companion report [Lane 90b]. For full details, see [Lane 90a].

5. Summary

This design space is directly usable as a notation for describing and comparing user interface system architectures. It should be useful for both the design and understanding of systems. The design rules provide a good starting point for the process of user interface structural design. As presented, the rules have been simplified too much to be capable of making subtle tradeoffs, but they can still help a designer to identify the better alternatives and to reject inappropriate structures. By reducing the mental effort needed to make the straightforward choices, these rules should free the designer to concentrate on the hard choices.

An automated form of the design rules has shown a substantial degree of agreement with the choices of human designers. One important implication of this result is that the design space provides considerable conceptual leverage: the space is "right" in the sense that using it makes choosing an appropriate design easier.

The design space and rules described here were based on an extensive survey of existing user interface systems [Lane 90a]. The space was formed by searching for classifications that brought systems with similar properties together. The rules were then prepared on the basis of observed correlations. This process can be compared to development of biological taxonomies through natural history: the biologist also surveys and classifies existing forms, then looks for explanatory theories.

At present there is no theoretical basis on which to argue that this design space is better or worse than a different set of dimensions that might be constructed to describe the same systems. The design space can be defended only on the grounds of practical utility: it seems to capture some useful design ideas and correlations. Further experience and research will no doubt improve this space, and someday a more theoretical, rigorous basis for creating design spaces may emerge.

Future work includes refining the design space and rules to cover lower-level choices, thus providing more detailed design advice. A full-scale attempt to automate the rules might produce a practical design aid. In the long term, we hope that this work can be generalized to yield principles of software architecture that hold beyond the domain of user interfaces.

Appendix A: The Design Space

This appendix provides a full description of the design space used in the experiment with automated rules. This space is probably somewhat different from what one would use in hand design work.

The design space contains twenty-five functional dimensions. Three to five alternatives are recognized in each of these dimensions. There are nineteen structural dimensions, each offering two to seven alternatives.

A.1. Functional Design Dimensions

We turn first to the functional design dimensions, which identify the requirements for a user interface system that most affect its structure. These dimensions fall into three groups:

- **External requirements:** Includes requirements of the particular applications, users, and I/O devices to be supported, as well as constraints imposed by the surrounding computer system.
- **Basic interactive behavior:** Includes the key decisions about user interface behavior that fundamentally influence internal structure.
- **Practical considerations:** Cover development cost considerations; primarily, the required degree of adaptability of the system.

A.1.1. External Requirements

A.1.1.1. Application Characteristics

The characteristics of the problem domain determine the features needed to provide an adequate user interface for a particular application or set of applications. A general-purpose user interface system may support more than one of the alternatives listed for any of these dimensions.

Primary output capability. What will be the system's main means of communicating information to its user? We classify the alternatives according to the type of data presented:

- **Text:** Displayed character strings.
- **Geometric graphics:** Images describable by geometric elements (lines, circles, etc). For example, engineering drawings.
- **General images:** Images not readily described by geometric elements, such as scanned photographs or bitmap artwork.
- **Voice:** Audible speech.
- **Audio:** Non-speech audible output, such as music or tonal signals.

Primary input capability. What is the system's main method of receiving information from its user?

- **Discrete selection:** Selection of one of a small set of alternatives; for example, selection from a menu, or a "yes/no" response.
- **Continuous selection:** Selection of a point in some continuum; for instance "pointing" to a point on a display surface, or manipulating a slider or control dial.
- **Text:** Textual data, usually typed on a keyboard; this is distinguished from discrete selection by a wider set of permissible inputs. (For instance, if the user is required to press `Y` or `N` to answer "yes" or "no," that is discrete selection via a keyboard; but entry of prose into a word processor, or names and addresses into a mailing list database, is textual input.)
- **Voice, discrete words:** Words are recognized individually, without use of grammar or context information.
- **Voice, connected speech:** Full-fledged speech recognition, using semantic context information to distinguish ambiguous words.

Command execution time. How long may the application program take to process a command, compared with the reaction time of a human user?

- **Short maximum time:** All commands can be executed in a short time, say a few tenths of a second.
- **Intermediate maximum, short average:** Most commands are executed in a short time, but some may take a bit longer, up to a couple of seconds.
- **Long maximum time:** Some or all commands may take a long time to execute so that the user will have a strong perception of waiting.

External event handling. Does the application program need to respond to external events, that is, events not originating in the user interface? If so, on what time scale?

- **No external events:** The application is uninfluenced by external events, or checks for them only as part of executing specific user commands. For example, a mail program might check for new mail, but only when an explicit command to do so is given. In this case, no support for external events is needed in the user interface.
- **Process events while waiting for input:** The application must handle external events, but response time requirements are not so stringent that it must interrupt processing of user commands. It is sufficient for the user interface to allow response to external events while waiting for input.
- **External events preempt user commands:** External event servicing has sufficiently high priority that user command execution must be interrupted when an external event occurs.

Error prevention importance. How important is prevention of user error, relative to other goals (such as speed of operation)?

- **High:** Error prevention is critical to the task (e.g., automated banking).

- **Medium:** Error prevention is of intermediate importance.
- **Low:** Error prevention is a minor issue.

A.1.1.2. User Needs

What features are needed for the intended user community? The dimensions affecting system structure are:

User help needs. How much user assistance is provided?

- **High:** Extensive assistance for novices is provided.
- **Medium:** Some guidance for novices is provided.
- **Low:** User interface is oriented towards expert users.

User experience variability. How much variability in experience is catered for?

- **High:** Different user interfaces are provided for novice and expert users.
- **Medium:** Minor changes in behavior are available for expert users.
- **Low:** No adaptation to different experience levels is provided.

User customizability. How much can a user modify the system's behavior? (We have in mind end users, not application developers.)

- **High:** User can add new commands and redefine commands (e.g., via a macro language), as well as modify user interface details.
- **Medium:** User can modify details of the user interface that do not affect semantics; for instance, change menu entry wording, default window sizes, colors, etc.
- **Low:** Little or no user customizability.

A.1.1.3. I/O Devices

What types of I/O devices will be used for communication with the user? The crucial aspects for system structure are:

Device class breadth. What range of I/O devices is supported by the user interface software? (We are interested here in the range of devices that are considered equivalent at some level of the software; for example, if two different displays are supported, they are probably equivalent at some level, but a display and a speaker would probably not be considered equivalent.)

- **Single device type:** Only a specific hardware type is permitted.
- **Semantically equivalent devices:** Devices with a fixed set of features are permitted; for example, 24x80 character terminals with cursor positioning and underlining capability. Any additional features possessed by a particular device are ignored. The means of invoking the required features may vary between devices.

- **Generic device definition:** A wide range of devices is permitted; for example, alphanumeric terminals of varying size with optional color and highlighting capabilities.

User interface adaptability across devices. How much change in user interface behavior may be required when changing to a different set of I/O devices?

- **None:** All aspects of behavior are the same across all supported devices.
- **Local behavior changes:** Only changes in small details of behavior across devices; for example, the appearance of menus.
- **Global behavior changes:** Major changes in surface user interface behavior; for example, a change in basic interface class (see below).
- **Application semantics changes:** Changes in underlying semantics of commands (e.g., continuous display of state versus display on command).

I/O device bandwidth. What data rate is needed to support the user interface I/O devices? (For devices with persistent state such as displays, use the burst rate needed for updates.)

- **High:** Kilobytes per second (e.g., high-resolution bitmap displays).
- **Medium:** Hundreds of bytes per second (e.g., alphanumeric terminals).
- **Low:** Tens of bytes per second (e.g., teletypes or small LED displays).

A.1.1.4. Computer System Environment

The surrounding computer system affects a user interface in several ways. The key issues are:

Strength of user interface conventions. How strong are the user interface conventions of the computer system?

- **High:** Extensive, well-defined standards which are generally followed (e.g., the Macintosh user interface guidelines [Apple 85]).
- **Medium:** Conventions exist but are incomplete and/or often violated (e.g., Unix conventions for command line syntax).
- **Low:** Little or no recognized common user interface behavior. (This is the situation for many stand-alone systems, such as automated store directories.)

Inter-application communication requirements. What kind of inter-application communication is supported *by the user interface*? ("Back door" communication such as data file exchange is not counted.)

- **None:** No communication at the user interface level.
- **Data exchange:** Via cut-and-paste or standardized I/O formats.
- **Program invokes program:** One program drives another, issuing commands

and interpreting responses. (Examples include Unix shell scripts and various macro languages.)

Inter-application protection requirements. To what extent does shared user interface software provide protection boundaries between different applications?

- **High:** User interface deals with multiple applications and must prevent undesirable interactions.
- **Medium:** User interface deals with multiple applications, but only weak protection is needed (e.g., applications are expected to cooperate).
- **Low:** No protection is needed (typically because user interface deals with only one application at a time).

Computer system organization. What is the overall organization of the computer system?

- **Uniprocessing:** A single application executes at a time.
- **Multiprocessing:** Multiple applications execute concurrently.
- **Distributed processing:** Network environment, with multiple CPUs and non-negligible communication costs.

Existing mechanisms for multiple threads of control. Does the operating system provide any mechanism(s) for multiple control threads?

- **Standard processes:** Independently scheduled entities with interprocess protection (typically, separate address spaces).
- **Lightweight processes:** Independently scheduled entities with no interprocess protection (shared address space).
- **Non-preemptive processes:** Processes without preemptive scheduling (must explicitly yield control); usually no interprocess protection.
- **Interrupt service routines:** Hardware-level event handling (a series of interrupt service routine executions can be viewed as a control thread).
- **None:** No system support for multiple control threads.

Processing power available for user interface. Is adequate processing power available for the user interface, or is it necessary to "cut corners" in the system design to achieve adequate response time?

- **High:** Plenty of processing power is available.
- **Medium:** Some care is needed to achieve adequate performance.
- **Low:** Must minimize resources used by user interface.

Designers usually make a rough judgment about available power at a fairly early stage in the design process, and this judgment colors many subsequent decisions. We include this dimension in the design space to make this judgment explicit.

A.1.2. Basic Interactive Behavior

This group of dimensions includes the key decisions about user interface behavior that fundamentally influence internal structure. Fortunately these are few; otherwise a single structure could not support a range of interaction styles.

Basic interface class. This dimension identifies the basic kind of interaction supported by the user interface system. (A general-purpose system might support more than one of these classes.) The design space uses a classification proposed by Shneiderman [Shneiderman 86]:

- **Menu selection:** Based on repeated selection from groups of alternatives; at each step, the alternatives are (or can be) displayed.
- **Form filling:** Based on entry (usually text entry) of values for a given set of variables.
- **Command language:** Based on an artificial, symbolic language; often allows extension through programming-language-like procedure definitions.
- **Natural language:** Based on (a subset of) a human language such as English.
- **Direct manipulation:** Based on direct graphical representation and incremental manipulation of the program's data.

It turns out that menu selection and form filling can be supported by similar system structures, but each of the other classes has unique requirements.

Degree of user control over dialog sequence. How much control does the user have over the sequence of interactions with the system?

- **High:** User controls dialog sequence (e.g., "modeless" dialog).
- **Medium:** User has some control over dialog.
- **Low:** Machine controls dialog sequence.

A.1.3. Practical Considerations

The remaining functional dimensions specify the required degree of adaptability of the system. In most cases a less adaptable system is cheaper to build. Yet a more adaptable system may repay its higher cost by supporting a wider class of applications. Another important consideration is that a system's adaptability affects its maintainability, and hence its lifespan.

It is useful to consider adaptability separately for application code and user interface code. The distinction disappears in single-purpose user interfaces, but is crucial for user interface systems that support multiple applications. We use the term *portability* for application code and *adaptability* for user interface code. This terminology is intended to connote the idea that we usually desire application code not to change when moving from one environment to another, while user interface support systems may well be modified to better adapt them to new environments. (Of course there are exceptions to this general rule.) Portability implies that the application is unaware of a change in environment, or at least can handle the change without being rewritten.

Application portability across I/O devices. What degree of portability across I/O devices is required for applications that will use the user interface software?

- **High:** Applications should be portable across devices of radically different types; for example, display versus speech output.
- **Medium:** Applications should be portable across devices of the same general class, but differing in detail; for example, bitmap displays of differing color capabilities.
- **Low:** Device independence is not a concern, or application changes are acceptable to support new devices.

Application portability across interaction styles. What degree of portability across user interface styles is required for applications that will use the user interface software?

- **High:** Applications should be portable across significantly different styles (e.g., command language versus menu-driven).
- **Medium:** Applications should be independent of minor stylistic variations (e.g., menu appearance).
- **Low:** User interface variability is not a concern, or application changes are acceptable when modifying the user interface.

Application portability across operating systems. What degree of portability across underlying computer systems is required for applications that will use the user interface software? (Primarily we are interested in operating system differences, though hardware differences may also be of interest.)

- **High:** Applications should be portable across significantly different machines and operating systems.
- **Medium:** Applications should be portable across related operating systems (e.g., portable to different versions of Unix).
- **Low:** System independence is not a concern.

User interface system adaptability across applications. How adaptable to different applications should the user interface software be?

- **High:** Useful across a wide range of applications.
- **Medium:** Useful for a group of closely related applications with similar interface needs.
- **Low:** Supports only a single application.

User interface system adaptability across interaction styles. How adaptable to different interaction styles should the user interface software be?

- **High:** Adaptable to a wide range of interface styles.
- **Medium:** Limited adaptability.
- **Low:** Imposes a specific interface style.

A user interface system may well be built to impose some stylistic decisions on applications; it is by no means the case that more flexibility is always better.

User interface system adaptability across operating systems. How adaptable to different computer systems should the user interface software be?

- **High:** Portable across significantly different machines and operating systems.
- **Medium:** Portable across related operating systems.
- **Low:** System independence is not a concern.

A.2. Structural Design Dimensions

We now turn to the structural dimensions, which represent the major decisions determining the overall structure of a user interface system. These dimensions fall into three major groups:

- **Division of functions and knowledge between modules:** How system functions are divided into modules, the interfaces between modules, and the information contained within each module.
- **Representation issues:** The data representations used within the system. We must consider both actual data, in the sense of values passing through the user interface, and *meta-data* that specifies the appearance and behavior of the user interface. Meta-data may exist explicitly in the system (for example, as a data structure describing the layout of a dialog window), or only implicitly.
- **Control flow, communication, and synchronization issues:** The dynamic behavior of the user interface code.

The structural design space presented here is a simplification of the complete design space discussed in [Lane 90a]. The simplification arises primarily from merging together decisions that proved to be closely correlated in practice. We will mention some of the omitted dimensions under the headings of the key dimensions with which they are associated.

A.2.1. Division of Functions and Knowledge

Under this heading, we consider how system functions are divided into modules, the interfaces between modules, and the information contained within each module.

The divisions of greatest interest are the divisions between application-specific code and shared user interface code on the one hand, and between device-specific code and shared user interface code on the other. We refer to these divisions as the *application interface* and *device interface*, respectively. (See Figure 2-1.)

Application interface abstraction level. The design space identifies six general classes of application interface. These classes can be most easily distinguished by their level of abstraction:

- **Monolithic program:** There is no separation between application-specific and shared code, hence no application interface (and no device interface, either).
- **Abstract device:** The shared code is simply a device driver, presenting an abstract device for manipulation by the application. The operations provided have specific physical interpretations (e.g., "draw line," but not "present menu"). Most aspects of interactive behavior are under the control of the application, although some local interactions may be handled by the shared code (e.g., character echoing and backspace handling in a keyboard/display driver). In this category, the application interface and device interface are the same.
- **Toolkit:** The shared code provides a library of interaction techniques (e.g., menu or scroll bar handlers). The application is responsible for selecting ap-

propriate toolkit elements and composing them into a complete interface; hence the shared code can control only local aspects of user interface style, with global behavior remaining under application control. The interaction between application and shared code is in terms of specific interactive techniques (e.g., "obtain menu selection"). The application can bypass the toolkit, reaching down to an underlying abstract device level, if it requires an interaction technique not provided by the toolkit. In particular, conversions between specialized application data types and their device-oriented representations are done by the application, accessing the underlying abstract device directly.

- **Interaction manager with fixed data types:** The shared code controls both local and global interaction sequences and stylistic decisions. Its interaction with the application is expressed in terms of abstract information transfers, such as "get command" or "present result" (notice that no particular external representation is implied). These abstract transfers use a fixed set of standard data types (e.g., integers, strings); the application must express its input and output in terms of the standard data types. Hence some aspects of the conversion between application internal data formats and user-visible representations remain in the application code.
- **Interaction manager with extensible data types:** As above, but the set of data types used for abstract communication can be extended. The application does so by specifying (in some notation) the input and output conversions required for the new data types. If properly used, this approach allows knowledge of the external representation to be separated from the main body of the application.
- **Extensible interaction manager:** Communication between the application and shared code is again in terms of abstract information transfers. The interaction manager provides extensive opportunities for application-specific customization. This is accomplished by supplying code that augments or overrides selected internal operations of the interaction manager. (Most existing systems of this class are coded in an object-oriented language, and the language's inheritance mechanism is used to control customization.) Usually a significant body of application-specific code customizes the interaction manager; this code is much more tightly coupled to the internal details of the interaction manager than is the case with clients of nonextensible interaction managers.

This classification turns out to be sufficient to predict most aspects of the application interface, including the division of user interface functions, the type and extent of application knowledge made available to the shared user interface code, and the kinds of data types used in communication. For instance, we have already suggested the division of local versus global control of interactive behavior that is typically found in each category.

Variability in device-dependent interface. The interface between device-dependent and device-independent code can be regarded as defining an **abstract device** for the device-independent code to manipulate. This dimension classifies abstract devices according to the degree of variability perceived by the device-independent code.

- **Ideal device:** The provided operations and their results are well specified in terms of an "ideal" device; the real device is expected to approximate the ideal behavior fairly closely.

- **Parameterized device:** A class of devices is covered, differing in specified parameters such as screen size, number of colors, number of mouse buttons, etc. The device-independent code can inquire about the parameter values for the particular device at hand, and adapt its behavior as necessary. Operations and their results are well specified, but depend on parameter values.

- **Device with variable operations:** A well-defined set of device operations exists, but the device-dependent code has considerable leeway in choosing how to implement the operations; device-independent code is discouraged from being closely concerned with the exact external behavior. Results of operations are thus not well specified.

- **Ad-hoc device:** In many real systems, the abstract device definition has developed in an ad-hoc fashion, and so it is not tightly specified; behavior varies from device to device. Applications therefore must confine themselves to a rather small set of device semantics if they wish to achieve portability, even though any particular implementation of the abstract device may provide many additional features.

The reader may wonder why there is no dimension that classifies abstract devices according to their basic functionality. Such a dimension might use categories like "bitmap display," "vector display," "alphanumeric display," "keyboard," "two-dimensional locator," etc. But there are a large number of such categories, with no obvious pattern. Moreover, much of the useful information has already been captured in other dimensions (device bandwidth, primary input and output capability). The simplified design space therefore provides no such dimension.

A.2.2. Representation of Information

Here we consider the representations used for user interface data. Since we are studying overall system structure, we are more interested in representations that are shared among modules than in those that are hidden within a single module.

Notation for user interface definition. This dimension classifies the techniques used for defining user interface appearance and behavior.

- **Implicit in shared user interface code:** Information buried within shared code. For example, the visual appearance of a menu might be implicit in the menu routines supplied by a toolkit.

- **Implicit in application code:** Information buried in the application and not readily available to shared user interface code.

- **External declarative notation:** A nonprocedural specification separate from the body of the application program, for example, a grammar or tabular specification. **Graphical specification** is an important special case, particularly useful for specification of visual appearance.

- **External procedural notation:** A procedural specification separate from the body of the application program; often cast in a specialized programming language.

- **Internal declarative notation:** A nonprocedural specification within the application program. This differs from an implicit representation in that it is available for use by the shared user interface code. Parameters supplied to user interface library routines often amount to an internal declarative notation. An example is a list of menu entries provided to a toolkit menu routine.
- **Internal procedural notation:** A procedural specification within the application program. This differs from an implicit representation in that it is available for use by the shared user interface code. A typical example is a status-inquiry or data transformation function that is provided for the user interface code to call.

Representation of semantic information. This dimension classifies the techniques used for defining application-specific semantic (as opposed to external appearance) information that is needed by the user interface. An example of such information is a range restriction on an input value.

- **Implicit:** Buried in the application, and not readily available to shared user interface code. For example, a range check carried out as part of command execution.
- **Declarative:** Expressed in a nonprocedural notation; for example, a form-filling package might allow range limits to be given in a table entry describing a numeric input field.
- **Procedural:** A procedural specification within the application program. This differs from an implicit representation in that it is available for use by the shared user interface code. For example, a validity checking subroutine might be provided for each input value.

The limited range of possibilities allowed by a declarative notation is more of a drawback here than it is for user interface definition. (Semantic information is inherently more variable across applications than surface user interface choices; were this not so, shared user interface behavior would be of no interest.) Procedural representations are therefore commonly used where shared code must have access to semantic information, while implicit representations are used where this can be avoided.

A.2.3. Control Flow, Communication, and Synchronization

Here we consider the dynamic behavior of the user interface code. As with the previous group of dimensions, we are mainly interested in inter-module communication.

Application control flow. Where does input processing occur in the application's flow of control?

- **Single input point:** The system contains an *event loop* that is the sole point at which user input is accepted; when an input event is received, it is processed; then control returns to the event loop to await the next input. Note that the event loop may be in either application or shared code.
- **Multiple input point:** Input is accepted at multiple points in the application's flow of control. (Usually, each such point can handle only a subset of the possible inputs, leading to modal interface behavior.)

This classification is a variation of the standard distinction between "internal control" (application calls user interface) and "external control" (user interface calls application) [Hayes 85, Tanner 83]. The standard terminology is unsatisfactory because the properties usually associated with external control actually apply to any system using an event loop, regardless of the direction of subroutine calls.

Treatment of asynchronous input. What happens to user input actions that occur while the application is busy?

- **Ignored:** Asynchronous input is ignored.
- **Queue before all processing:** Input events are queued, but no processing is done (and hence no feedback occurs) until the application is ready for input.
- **Partial processing, simple queue:** Some fast processing is done to provide feedback; then events are queued for the application in a first-in-first-out queue.
- **Partial processing, complex queue:** As above, but the queue may not be strictly FIFO; for instance, "abort" commands may be delivered first, or may flush the queue.

Note that the first two of these alternatives correspond to no fast input processing, while the second two describe systems which have some type of fast input processing.

Fast input processing. Is user input processed before the application is ready to receive it? If so, how flexible is this processing?

- **No fast processing:** Everything is synchronous with the application.
- **Fixed behavior:** Some processing and feedback is done asynchronously; the nature of the asynchronous processing is not alterable by the application. (Example: input echoing and editing in older time-sharing systems.)
- **Parameterized behavior:** Application-specific code can set limited parameters for the behavior of the asynchronous processing. For example, in some window systems, different cursor shapes can be established for different parts of an application’s window. Shape changes are then handled automatically by the cursor tracking code.
- **Application-dependent behavior:** Application-specific code can be executed during fast processing. For example, an application-specific routine might be used to draw rubber-band feedback images during dragging.

The more flexible alternatives in this dimension carry increasing risk of synchronization problems. (A simple example is that typed-ahead characters may be echoed twice or not at all when switching between asynchronous echoing and application-driven echoing.) Communication costs can also be a problem for the last alternative.

Number of control threads. How many control threads exist in the application and user interface?

- **Single thread of control.**

- **One user interface thread and one application thread.**
- **Multiple user interface threads and one application thread.**
- **One user interface thread and multiple application threads.**
- **Multiple user interface threads and multiple application threads.**

Multiple threads are useful for dealing with external events or logically independent concurrent dialogues (e.g., multiple input devices). The one-plus-one-thread choice is particularly simple and helpful for decoupling application processing (including external event handling) from user interface logic.

Control thread mechanism. What mechanism, if any, is used to support multiple control threads?

- **None:** Only a single logical control thread is used.
- **Standard processes:** Independently scheduled entities with interprocess protection (typically, separate address spaces).
- **Lightweight processes:** Independently scheduled entities within a shared address space.
- **Non-preemptive processes:** Processes without preemptive scheduling (must explicitly yield control), usually in a shared address space.
- **Event handlers:** Pseudo-processes which are invoked via a series of subroutine calls; each such call must return before another event handler process can be executed.
- **Interrupt service routines:** Hardware-level event handling; a series of interrupt service routine executions form a control thread, but one with restricted control flow and communication abilities. Unlike simple event handlers, preemptive scheduling is available.

Application communication grain size. How frequently does communication occur between application and shared user interface code?

- **Fine grain:** Roughly once per user input event; the application is closely coupled to user actions, and typically participates in feedback generation.
- **Coarse grain:** Roughly once per complete command; the application is decoupled from user actions and feedback generation.

Either of these approaches may be preferable, depending on the desired extent of application involvement in user interface details.

Device communication grain size. How frequently does communication occur between device-independent and device-dependent code?

- **Fine grain:** Roughly once per physical input event; the device-independent code is involved in generating short-term feedback displays.

- **Coarse grain:** Roughly once per logical interaction; the device-independent code is not involved in short-term feedback generation.

Basis of communication. Does communication between modules depend on shared state or on events, or both? (An *event* is a transfer of information occuring at a discrete time, for example, via a procedure call or message.)

- **Events:** There is no shared state; all communication relies on events.
- **Pure state:** Communication is strictly via shared state; the recipient must repeatedly inspect the state variables to detect changes.
- **State with hints:** Communication is via shared state, but the recipient is actively informed of changes via an event mechanism; hence polling of the state is not required. However, the recipient could ignore the events and reconstruct all necessary information from the shared state; so the events are efficiency hints rather than essential information.
- **State plus events:** Both shared state and events are used; the events are crucial because they provide information not available from state monitoring.

It is possible for different bases of communication to be used at the application and device interfaces, but this is rare. It is fairly common to have different bases of communication for input and output; hence the design space provides separate dimensions for input and output communication basis.

Event mechanisms. Unless pure-state communication is used, a mechanism must be provided to pass events between modules. We classify event mechanisms thus:

- **None:** No events are used (pure state communication).
- **Direct procedure call:** Standard procedure-call mechanism. (We include "remote procedure call" mechanisms, so long as the recipient code is directly named.)
- **Indirect procedure call:** Procedure call in which the recipient code is not completely specified by the calling code, but is dynamically determined; procedure pointers and object-oriented method calls are typical examples.
- **Asynchronous message:** The event is passed from one control thread to another, with the sender not waiting for receipt.
- **Synchronous message:** The event is passed from one control thread to another, with the sender blocked until the receiver accepts the message (and computes a reply, usually). This differs from a remote procedure call in that the receiver is a separate control thread that exists before and after the rendezvous.

The procedure call mechanisms are used for communication within a control thread, the message mechanisms for communication across threads. Indirect procedure calls provide extra separation at slightly higher cost. Synchronous message mechanisms are somewhat cheaper to implement than asynchronous ones (for instance, message buffering can be avoided), but they may create synchronization problems by increasing timing dependencies between control threads.

It is common to have different event mechanisms for input and output, and also to have different mechanisms at the application and device interfaces. Hence the design space provides four event-mechanism dimensions, one each for application input, application output, device input, and device output.

Application separation mechanism. How strongly are the application and shared user interface code separated?

- **Programming convention:** No mechanism exists to enforce separation.
- **Visibility rules:** A programming language mechanism such as separate name spaces. Protection strength depends on whether the language is secure against errors (such as dangling pointers).
- **Hardware separation:** A hardware mechanism, typically separate address spaces. The shared user interface code is reliably protected against programming errors in the application (and vice versa).
- **Network link:** In addition to providing hardware separation, the communication protocol allows for cross-machine communication; data representation differences between application and user interface code are supported. An example is the support for varying byte order in the X Window protocol.

These choices provide a tradeoff of security against cost of communication. The availability of suitable mechanisms is also a consideration; many small machines do not provide hardware protection mechanisms.

Device separation mechanism. How strongly are the device-dependent and device-independent layers separated?

The classification is the same as for the previous dimension.

The data volume and frequency of communication are usually higher here than at the application interface, so the cost of communication is a greater concern. Thus a different choice is often appropriate.

Appendix B: The Design Rules

This appendix presents some simple "rules of thumb" that help a designer of user interface software to select a system architecture. These rules are not meant to replace good design judgment, but rather to codify and speed up the routine parts of system design. The rules let the designer make quick decisions about aspects of system structure for which there is a clearly superior alternative, and they focus attention on the most likely choices in cases where more subtle judgment is necessary.

We discuss the design dimensions in the order in which a designer might consider them while creating a design. For each dimension, we present a listing of the considerations that may favor or disfavor each alternative, and some summary rules-of-thumb for selecting one alternative. Again we emphasize that these rules must be augmented by the designer's judgment: typically, the designer must resolve conflicting suggestions by judging the relative importance of different functional requirements.

Space limitations prohibit any attempt to provide justifications of these observations and rules. Supporting arguments can be found in [Lane 90a].

B.1. Basic Division of Functions

The designer's first order of business should be to define the overall division of a system into device-specific, shared user interface, and application-specific parts. We view this as a problem of specifying two interfaces: the *application interface* between application-specific and shared code, and the *device interface* between device-specific and shared code.

B.1.1. Application Interface

Application interface abstraction level. The design space identifies six general classes of application interface. In order of increasing level of abstraction in communications, they are:

- **Monolithic program:** This is an appropriate solution in small, specialized systems where the application needs considerable control over UI details and/or little processing power is available. (Video games are a typical example.) This approach should not be chosen if there are any strong flexibility requirements (user customizability, I/O device variability, or UI style flexibility). The approach handles direct manipulation interfaces well, but application development effort will be high.
- **Abstract device:** This approach is recommended when application portability is wanted across a limited set of devices, but most control of the user interface is to remain in the hands of the application program. Thus it is not a good choice when application portability across UI styles is a strong requirement. It is best not to attempt to support a very wide range of I/O devices with this approach; the result will be either excess development effort for applications (too many cases to handle) or loss of control over UI details (if the driver hides too

many details). The characteristics of this approach are heavily influenced by the handling of abstract device variability, which is discussed in Section B.1.2.

- **Toolkit:** Toolkits provide a significant savings of application development effort, and yet retain UI system flexibility since the application remains "in charge" and can bypass the toolkit when necessary. By the same token, the application remains coupled to the user interface. Therefore, this approach is recommended when a moderate degree of flexibility is wanted. This approach is the minimum level of abstraction to use when a standardized UI style is to be implemented, because standard components (e.g., menus) can be handled by toolkit routines rather than reimplemented by each application.

- **Interaction manager (IM):** An IM is a good choice when application portability (across devices or styles) is a strong requirement, because it provides a strong separation between UI behavior and the application program. A high degree of user customizability can also be supported. However, supporting direct manipulation interfaces is difficult because the application cannot supply semantic feedback. An IM is useful for enforcing standardized UI behavior, since it gives the application program the least control over UI details of any alternative. An IM is especially appropriate in network environments, because the IM can be physically separated from the application with low communication costs.

- **Interaction manager with extensible data types:** Some IMs provide the capability to extend the set of data types used for application/IM communication. This option allows representation conversion to be fully separated from the main body of the application, but it does not do much to solve the semantic feedback problem. Hence it provides only a small increment in flexibility.

- **Extensible interaction manager:** This is accomplished by supplying code that augments or overrides selected internal operations of the IM. An extensible IM can provide as much support as a regular IM for standardized styles of user interface. But it can be used for a wider class of interfaces---including direct manipulation---by taking advantage of its customization capability. This approach provides the most flexibility for meeting user customizability, I/O device variability, and UI style requirements. But it requires substantial processing power, and the level of initial investment (for both UI system development and application developer training) is higher than for any other alternative. Moreover, care is needed to realize the potential flexibility benefits; since application-specific customization code sees a relatively low level of abstraction, it is easy to destroy the logical separation between application and user interface system.

The benefit to be gained from building anything more complex than an abstract device system depends heavily on the degree of standardization of UI behavior---that is, the strength of the UI conventions in the system environment. The more that such conventions limit the range of UI behavior, the more functionality can be put into a toolkit or IM, and the less need there is for an application to override standard behavior. Thus increasing strength of conventions tilts the balance first towards toolkits and extensible IMs, then towards nonextensible IMs.

A nonextensible IM may be the best choice when application portability and development cost are paramount, as it provides the most insulation of the application from UI details. Its limited range of UI styles is a necessary price; at least with present technology, direct manipulation systems cannot be built without significant application involvement in the user interface, which compromises both portability and cost.

B.1.2. Device Interface

The interface between device-independent and device-specific code can be regarded as defining an **abstract device** for the device-independent code to manipulate. The details of an abstract device vary greatly across I/O media, but some general statements can be made about the precision with which the abstract device is specified.

Abstract device variability. This dimension classifies abstract devices according to the degree of variability perceived by the device-independent code.

- **Ideal device:** In this approach, all questions of device variability are hidden from software above the device driver level, so application portability is high. This approach is most useful where the real devices deviate only slightly from the ideal model, or at least not in ways that require rethinking of UI behavior. The ideal-device approach is not appropriate if any major changes in UI behavior may be needed to cope with differences between devices; therefore it cannot cover as wide a range of actual devices as the other two approaches.

- **Parameterized device:** This approach allows a wide range of I/O devices to be accommodated, and it permits substantial changes in UI behavior across devices. The advantage is that application-specific code has both more knowledge of acceptable tradeoffs and more flexibility in changing its behavior than is possible for a device driver. The drawback is that device-independent code may have to perform complex case analysis in order to handle the full range of supported devices. If this must be done in each application, the cost is high and there is a great risk that programmers will omit support for some devices. (To reduce this temptation, it is best to design a parameterized model to have just a few well-defined levels of capability, so as to reduce the number of cases to be considered.) This approach should not be used if it is not necessary to support a wide range of I/O devices, as then its high cost is not repaid. Less obviously, it should not be used when high application portability across I/O devices is crucial (unless the application is insulated from the abstract device by an IM layer); the risk of applications failing to cover the full range of parameter variation is too great. A final drawback is that substantial processing power is likely to be needed to handle extensive runtime case analysis.

- **Device with variable operations:** This approach works best when the device operations are chosen at a level of abstraction high enough to give the device driver considerable freedom of choice. Hence the device-independent code must be willing to give up much control of UI details. This restriction means that direct manipulation (with its heavy dependence on semantically-controlled feedback) is not well supported. Furthermore, only local changes in interface behavior can be handled at the device driver level; changes in basic interface class or application semantics cannot be supported. When these restrictions are acceptable, this approach can support a very wide range of devices with little impact on device-independent code. Its costs in processing power are low, since runtime case analysis need not be performed.

- **Ad-hoc device:** This approach is hardly ever appropriate for new designs. It is found principally in systems that have evolved from simpler beginnings.

In systems where little or no variation in I/O devices is expected, one may as well specify an ideal device model (tailoring it closely to the real devices); this incurs no runtime cost and provides a well-defined picture of what is required if more devices need to be supported later. When a moderate or wide range of I/O devices must be supported, the key question is what types of UI behavior changes are needed across devices. Parameterization is essential if global changes are needed, as the device driver cannot handle such changes alone. Moderate local changes are well served by the variable-operations method, if its drawbacks are tolerable; otherwise parameterization is preferred. An ideal device approach may still be usable if only small, local changes in behavior are needed.

It is possible to support multiple tradeoffs between handling device adaptation in the device driver and handling it in the application: simple applications can rely on device-specific UI decisions made in the driver, while more complex ones can make their own choices. This amounts to a combination of the variable-operations and parameterized-device approaches. Obviously, to make this work well, great care is needed in defining the device operations and parameters.

Selecting the functions to be provided in an abstract device model is a complex task. A poorly chosen model may limit portability and/or cause performance problems due to mismatches between its properties and specific real devices. Unfortunately, good designs seem very dependent on properties of the particular I/O medium; few general design principles have emerged. We can suggest some rules of thumb based on the chosen degree of variability. When using an ideal or parameterized-device approach, it is probably best to minimize the amount of user interface functionality (i.e., representation conversion, sequence control, user assistance, and state maintenance) placed in the device driver. The variable-operations approach, in contrast, gains its power precisely by moving significant user interface decisions into the device driver. The trick here is to choose a coherent set of decisions that are not tightly coupled to those remaining in higher level software. (Some of the problems with GKS input devices are due to failure to maintain this separation [Rosenthal 81].)

B.2. Representation Issues

After defining the major system components and allocating functionality among them, the next order of business is selecting data representations to be used within the system. We must consider both actual data, in the sense of values passing through the user interface, and *meta-data* that specifies the appearance and behavior of the user interface. Meta-data may exist explicitly in the system (for example, as a data structure describing the layout of a dialog window), or only implicitly. We further subdivide meta-data according to whether it bears on "surface" UI details or on deeper questions of application semantics.

B.2.1. User Interface Definition

Notation for user interface definition. Here we consider the means of defining UI appearance and behavior: the meta-data that describes surface details. We classify notations for UI definition as follows:

- **Implicit in shared user interface code:** This is simple and efficient; it is the appropriate choice for UI behavior that is fixed by the support software. In systems where strong UI conventions exist, quite a lot of the definition can reasonably be represented this way. It should be avoided when the user interface system is to be adaptable across a wide range of UI styles, or when user customizability is important.
- **Implicit in application code:** This is the traditional approach that most UI researchers have tried to move away from. But it will never be eliminated entirely since it, too, is simple and efficient. It is most appropriate where the application is already tightly involved in the user interface, for example, in handling semantic feedback in direct manipulation systems. It should be avoided when application portability (across I/O devices or UI styles) or user customizability is important.
- **External declarative notation:** Declarative representations in general provide the least flexibility of interface design, but are the easiest to use. External declarative notations are particularly well suited to supporting user customization and to use by nonprogramming UI experts. Use of an external notation helps keep the main application code portable across UI styles and I/O devices, but only if the notation is flexible enough to specify all the required variations by itself. Processing power requirements can be high, unless the notation can be precompiled in some way. **Graphical specification** is a special case of external declarative notation; graphical methods are particularly appropriate for specification of visual appearance.
- **Internal declarative notation:** From the application programmer’s viewpoint this is nearly as easy to use as external declarative notation, and it requires much less supporting mechanism; however, it makes user customization much more difficult.
- **External procedural notation:** Procedural notations are more flexible than declarative ones, but are harder to use. User-accessible procedural mechanisms, such as macro definition capability or the programming language of EMACS-like editors, provide very powerful customization possibilities for sophisticated users. Use of an external notation helps keep the main application code portable across UI styles and I/O devices. Substantial processing power may be needed, depending on the efficiency of the mechanism that executes the notation. Also, an external notation by definition has limited access to the state of the application program, which may restrict its capability.
- **Internal procedural notation:** This is the most commonly used notation for customization of extensible interaction managers. It provides an efficient and

flexible notation, but is not accessible to the end user, and so is useless for user customization. It is particularly useful for handling application-specific feedback in direct manipulation interfaces, since it has both adequate flexibility and efficient access to application semantics. This approach is not favored when application portability is a strong requirement.

Typically, several kinds of notation are used in a user interface system. Almost always there are some instances of both kinds of implicit notation, and one or more of the others is often used as well. The crucial question is thus which aspects of UI behavior should be described in which kinds of notation. The best indicators of the appropriate class of notation are the required degrees of flexibility and efficiency.

A good rule of thumb is that declarative notations are appropriate for static information or restricted choices, such as the layout of a display or the selection of one of several predefined behaviors. Procedural notations are a better choice for description of dynamic behavior, because presently available declarative methods aren't sufficiently flexible. In either case, an external notation should be used when user customization is required; otherwise an internal notation is simpler and more efficient. Implicit representation should be used only when efficiency is crucial or the probability of change is low.

B.2.2. Application Semantic Information

Representation of semantic information. This dimension classifies the techniques used for defining application-specific semantic (as opposed to external appearance) information that is needed by the user interface. An example of such information is range restrictions on an input value. The classes are:

- **Implicit.**
- **Declarative.**
- **Procedural.**

The limited range of possibilities allowed by a declarative notation is more of a drawback here than it is for user interface definition. (Semantic information is inherently more variable across applications than surface user interface choices; were this not so, shared UI behavior would be of no interest.) Procedural representations are therefore the best bet where shared code must have access to semantic information, while implicit representations are usually used otherwise. In cases where only a limited number of alternatives are likely to be needed, declarative representations are recommended for ease of use.

Natural language interfaces have special requirements: a great deal of semantic information must be explicitly represented for use in disambiguating sentences. Both declarative and procedural techniques are commonly used.

B.2.3. Representation of Data Values

It turns out that the application interface class is usually sufficient to predict the kinds of data types passed between modules, so the design space does not include a separate dimension for this issue.

The lowest application interface abstraction levels rely on device-related data types, such as bitmaps or other image representations for displays, or keystroke sequences for keyboards. Toolkit systems introduce data types for user interface constructs such as menus or scroll bars. Interaction managers use "internal" data types that might be directly used within application computations, such as integer or floating-point values. Simple IMs use a fixed set of standard internal types, while extensible IMs can be extended to communicate in terms of application-specific internal data types.

As a rule of thumb, application-related data types should be used in preference to device-related data types. For example, integer or Boolean values are preferred to equivalent character strings or bitmaps. This rule encourages moving representation conversions into the user interface code.

B.3. Control Flow and Synchronization

We turn now to questions of control flow: what are the control relationships between the system components, and how are sequences of events synchronized?

It is convenient to visualize control flow in terms of logical *control threads*. A control thread is an entity capable of independently performing computations and waiting for events to occur. We use this term in place of "process" because we do not want to restrict the notion to standard operating-system-supplied processes. (Section B.5.2 lists numerous mechanisms that can support the logical notion of a control thread, possibly with some restrictions in thread structure or event response time.)

Application control flow. Our most basic control flow dimension is a variation of the standard distinction between "internal control" and "external control" [Hayes 85]. We prefer to define the categories as:

- **Single input point.**
- **Multiple input point.**

A single input point is appropriate for creating "modeless" interfaces. Even with a moded interface, building the application in single-input-point style can be helpful, since it serves to decouple the application from details of user interface sequencing. Hence high requirements for application portability or user customizability favor single input point control flow. The major advantage of multiple input point flow is that application actions need not be atomic with respect to user interaction. Generally, multiple input points should be used only if this is an essential feature.

A single input point is also desirable when external events are to be handled while waiting for user input; then there is only one point at which to worry about external events.

Number of control threads. This dimension counts the control threads:

- **Single thread:** This approach is adequate for simple systems, particularly if single input point control flow can be used (i.e., "external control" of the application is sufficient). It is usually not appropriate when external event handling is important, nor when long command execution times occur.
- **One UI thread and one application thread:** This alternative is very popular since it decouples user interface control flow from the application. Two threads are sufficient to allow user interface operations to execute concurrently with the application. On the user interface side, this allows user input to be processed and feedback displays to be updated while commands are being executed. On the application side, external events can be handled without impeding user interface response, and the application is made more independent of user interface event sequencing. The cost of providing a multiple-control-thread mechanism is the major drawback to using this approach. An existing control thread mechanism may be usable, depending on the cost of communication between threads.
- **Multiple UI threads:** Multiple UI threads simplify dealing with logically independent parallel interactions. These occur in modeless interfaces and when multiple input devices are used. An inexpensive thread mechanism is necessary to make this a reasonable approach.
- **Multiple application threads:** Multiple application threads may be useful for dealing with external events. Some systems use them to control cancellation of user commands.

If an inexpensive control thread mechanism is available, the two-thread approach should be used for all but the very simplest user interfaces. The tradeoff point changes if one must build one's own thread mechanism, although a simplified mechanism may be adequate. If independent concurrent sequences of events must be dealt with, explicit use of multiple threads is nearly always the right choice. Even with a restrictive thread mechanism, this will be cleaner and more reliable than ad hoc solutions.

If external event handling is required to preempt user command execution, a thread mechanism that provides preemptive scheduling is very desirable. Without one, it will be necessary to poll for external events during command execution; this is feasible, but inefficient and error-prone.

Treatment of asynchronous input. The user interface must have a strategy for handling asynchronous input events (events that occur while the application is computing). The standard approaches are:

- **Ignore asynchronous input:** Often the simplest approach to implement, and it has some advantages in terms of simplicity of UI behavior. It is usually not appropriate if commands may take a long time to execute.

- **Queue before all processing:** A satisfactory solution if events do not remain in the queue for long. Otherwise, the lack of feedback is a serious human factors shortcoming. Hence this approach is also inappropriate if commands may take a long time to execute; but it is usually the best solution for short or intermediate command times.
- **Partial processing with queuing:** Provides flexibility, but requires multiple control threads and introduces synchronization concerns. Hence it should be avoided unless necessary (i.e., unless there are long commands).

Fast input processing. When partial processing is provided, the variability of behavior of the fast processing is an important issue. This may be:

- **Fixed behavior:** Simple and has no synchronization problems, but is obviously inflexible. It is sufficient if user interface system adaptability is not a strong requirement.
- **Parameterized behavior:** Recommended in most cases, because the parameter semantics can be defined to minimize synchronization problems. (In particular, one should be wary of parameters that will be changed "on the fly" when already-processed input may be pending.)
- **Application-dependent behavior:** Should be used only if user interface system adaptability requirements are so high as to make it mandatory. Use of application-supplied fast processing routines reduces application portability and creates significant synchronization concerns.

The more flexible alternatives in this dimension carry increasing risk of synchronization problems. (A simple example is that typed-ahead characters may be echoed twice or not at all when switching between asynchronous echoing and application-driven echoing.) Communication costs can also be a problem for the last alternative. In general, one should use the least flexible method possible.

Application communication grain size. How frequently does communication occur between application and shared user interface code?

- **Coarse grain:** This is suitable when the application need not be involved in the details of UI interactions.
- **Fine grain:** This is most likely to be required in direct manipulation interfaces. Communication costs and application portability are sacrificed, so this alternative should not be used unless necessary.

Coarse-grained communication should be used if the application has long-running commands or external events to cope with, since then one cannot rely on it to provide feedback promptly.

It is also possible to distinguish between coarse-grained and fine-grained communication at the device interface. In coarse-grained device communication, the device-specific code handles feedback for entire sequences of input events; while with fine-grained device communication, feedback is handled at higher levels. As a rule of thumb, fine-grained device communication is preferable. Coarse-grained communication may be acceptable if substantial control of the user interface is to be put in the device driver level; this is associated with the device interface classification of abstract devices with variable operations.

B.4. Matters of State

The system architecture should explicitly recognize state information, whether hidden within one module or shared between modules. Shared state is a useful vehicle for communication. Shared or not, the existence of persistent state is a key aspect of system semantics and an important basis for performance optimization.

B.4.1. Representation of Interface State

How to represent the state of the user interface is a very general question. Our rules of thumb address only a small part of it, to wit: whether to retain intermediate representations of output (such as display lists or cached bitmaps). Intermediate representations take extra work to maintain, but can provide valuable benefits. We recommend maintaining an intermediate output representation when (1) the output device can usefully be treated as having a state (not true for audio output, for instance); (2) recalculating the output device's state from scratch (from underlying application state) is expensive; and (3) partial or incremental updates are common. Under these conditions the performance gain is worth the extra trouble.

Intermediate output representations are also important for handling reference interpretation (e.g., deducing that a mouse click represents a menu element selection). This may justify maintaining an intermediate representation even when display update savings are not significant. A partial representation (e.g., just menu coordinates) may be enough for this purpose.

B.4.2. Communication via Shared State

Basis of communication. Communication between modules may depend on shared state or on events, or both. (An *event* is a transfer of information occuring at a discrete time, for example via a procedure call or message. Communication through shared state variables is significantly different because the recipient need not use information in the same order in which it is sent.)

- **Events.**
- **Pure state.**
- **State with hints.**
- **State plus events.**

State-based communication can be recommended for driving devices that exhibit persistent state, such as displays. The use of explicit state is a natural way of formalizing the maintenance of intermediate representations of output (see above). However, event-based communication is more appropriate for devices that have no useful characterization of state.

The hybrid communication forms which combine events with shared state allow improved performance at the price of increased complexity. As a rule of thumb, pure state systems are simpler and less efficient than pure event systems, which in turn are simpler and less efficient than hybrid systems.

The major drawback to state-based communication is that it requires efficient access to shared storage. This may not be available in multi-process systems, especially when communication across network links is involved. Synchronization issues must also be considered if multiple threads access the shared state.

B.5. Mechanisms

The final group of dimensions concern the mechanisms used to implement communication and control flow. The classifications used here are the lowest level of detail that can reasonably be described as part of the system architecture. But these issues are indeed part of system architecture, because they have strong implications for questions that we have already discussed.

B.5.1. Communication Mechanisms

Event mechanisms. A pure state-based system has no events and so needs no event communication mechanism. The other three classes of communication require a mechanism to pass events between modules. For communication within a single control thread, the alternatives are:

- **Direct procedure call.**
- **Indirect procedure call.**

Indirect calls provide useful separation between the communicating modules. If the chosen programming language has a natural mechanism for representing indirect calls, they are usually well worth the small runtime cost; but otherwise the difficulty of using indirect calls may outweigh their value.

For communication between control threads, the alternatives are:

- **Asynchronous message.**
- **Synchronous message.**

Asynchronous messages are often superior since they reduce synchronization problems and can be batched to reduce overhead. Synchronous messages have simpler semantics and sometimes can be implemented more easily (e.g., message buffers may not be needed). If a message mechanism is already available, one should probably use it by default; otherwise asynchronous messages seem better suited to most UI purposes.

Separation mechanisms. A separation mechanism isolates software components while still permitting communication. We recognize four classes:

- **Programming convention:** This approach provides very weak protection, but it is flexible and incurs no runtime cost. This is a reasonable choice for communication between closely related components, or when the system components are automatically generated (and thus less prone to human coding error).
- **Visibility rules:** This type of mechanism is quite flexible, since the programmer can choose what to export or hide. The runtime cost is small: at most, a procedure call is needed to cross a protection boundary. In many programming languages the protection is not secure against runtime errors.
- **Hardware separation:** Security is strong, but the cost of communicating across the protection boundary is high---often several orders of magnitude more expensive than a procedure call. This is an appropriate choice when it is important to ensure security, for example in a window manager that serves multiple applications. This approach may also be necessary for communication between modules coded in different programming languages. An important aspect of hardware separation is that most current operating systems associate these protection boundaries with processes; hence division of the user interface system into protectable entities must be considered jointly with control flow and synchronization concerns.
- **Network link:** The communicating parties can exist on nonidentical machines. The cost of communication in such a case is inherently high, but is worthwhile in distributed environments.

Generally, visibility rules are the minimum separation that should be used between application and user interface code. Stronger separation mechanisms should be used only where there are system considerations that justify their cost. The major considerations that may justify a stronger mechanism are (1) the need for a shared user interface system to protect itself against errors in any one application; (2) use of a system-provided process mechanism that forces hardware separation; or (3) the desire to distribute system components across machines in a network.

Separation will also exist between the shared user interface code and the device-specific code. This may or may not use the same class of mechanism as is used at the application interface. In most cases visibility rules are sufficient; the main exception is to permit distribution across a network.

B.5.2. Control Flow Mechanisms

Control thread mechanism. Among the many ways to provide the abstract notion of a control thread are:

- **Standard processes:** These provide security against other processes, but interprocess communication is relatively expensive. For a user interface system, security may or may not be a concern, while communication costs are almost always a major concern. If the operating system already provides processes, not having to implement one's own process mechanism is an important advantage. In network environments, standard processes are usually the only kind that can be executed on different machines.
- **Lightweight processes:** These are suitable only for mutually trusting processes due to lack of security; but often that is not a problem for user interface systems. The benefit is substantially reduced cost of communication, especially for use of shared variables. Few operating systems provide lightweight processes, and building one's own lightweight process mechanism can be difficult.
- **Non-preemptive processes:** These are relatively simple to implement since no preemption mechanism is needed. Synchronization can be achieved merely by not yielding the processor, although explicit interlocks are safer. The major drawback is that response to I/O devices can be slow, and response time is hard to control.
- **Interrupt service routines:** These provide a simple preemptive scheduling mechanism. The control flow and communication patterns of ISR-implemented processes are very restricted, but they are useful for ensuring fast response to I/O devices. ISRs are highly machine-dependent, and may not be available to unprivileged programs.
- **Event handlers:** The main advantage of this method is that it requires virtually no support mechanism. The key disadvantages are the control flow restrictions, which are comparable to ISRs, and the lack of fast response, which is comparable to non-preemptive processes.

Of these, the most commonly useful alternatives are standard processes, lightweight processes, and event handlers; the others are appropriate only in special cases. For most user interface work, lightweight processes are very appropriate if available. Standard processes should be used when protection considerations warrant, and in network environments where it may be useful to put the processes on separate machines. If these conditions do not apply, event handlers are the best choice when their response time limitations are acceptable; otherwise it is probably best to invest in building a lightweight process mechanism.

References

[Adobe 85] *PostScript Language Reference Manual*
Adobe Systems, Inc., 1985.
Published by Addison-Wesley.

[Apple 85] *Inside Macintosh*
Apple Computer, Inc., 1985.
Published by Addison-Wesley. Volumes I-IV.

[Bell 71] C. Gordon Bell and Allen Newell.
Computer Structures: Readings and Examples.
McGraw-Hill, New York, 1971.

[Borenstein 88] Nathaniel S. Borenstein and James Gosling.
UNIX Emacs: A Retrospective (Lessons for Flexible System Design).
In *Proceedings of Symposium on User Interface Software*, pages 95-101. ACM SIGGRAPH, Banff, Alberta, Canada, October 1988.

[Dance 87] John R. Dance, Tamar E. Granor, Ralph D. Hill, Scott E. Hudson, Jon Meads, Brad A. Myers, and Andrew Schulert.
The Run-time Structure of UIMS-Supported Applications.
Computer Graphics 21(2):97-101, April 1987.
ACM SIGGRAPH Workshop on Software Tools for User Interface Management.

[Green 86] Mark Green.
A Survey of Three Dialogue Models.
ACM Transactions on Graphics 5(3):244-275, July 1986.

[Hartson 89] H. Rex Hartson and Deborah Hix.
Human-Computer Interface Development: Concepts and Systems for Its Management.
ACM Computing Surveys 21(1):5-92, March 1989.

[Hayes 85] Philip J. Hayes, Pedro A. Szekely, and Richard A. Lerner.
Design Alternatives for User Interface Management Systems Based on Experience with Cousin.
In *Proceedings of CHI '85: Human Factors in Computing Systems*, pages 169-175. ACM SIGCHI, 1985.

[Knuth 73] Donald E. Knuth.
The Art of Computer Programming. Volume 1: *Fundamental Algorithms.*
Addison-Wesley, Reading, MA, 1973.

[Lane 90a] Thomas G. Lane.
User Interface Software Structures.
PhD thesis, Carnegie Mellon University, May 1990.
CMU School of Computer Science Technical Report CMU-CS-90-101. Also available as Software Engineering Institute Special Report CMU/SEI-90-SR-13.

[Lane 90b] Thomas G. Lane.
Studying Software Architecture through Design Spaces and Rules.
Technical Report CMU/SEI-90-TR-18, Carnegie Mellon University Software Engineering Institute, October 1990.
Also available as CMU School of Computer Science Technical Report CMU-CS-90-175.

[Lantz 87] Keith A. Lantz, Peter P. Tanner, Carl Binding, Kuan-Tsae Huang, and Andrew Dwelly.
Reference Models, Window Systems, and Concurrency.
Computer Graphics 21(2):87-97, April 1987.
ACM SIGGRAPH Workshop on Software Tools for User Interface Management.

[Myers 89] Brad A. Myers.
User-Interface Tools: Introduction and Survey.
IEEE Software 6(1):15-23, January 1989.
An earlier version was published as Carnegie Mellon University Technical Report CMU-CS-88-107, January, 1988.

[Perry 84] Robert H. Perry, Don W. Green, and James O. Maloney.
Perry's Chemical Engineers' Handbook.
McGraw-Hill, New York, 1984.

[Reid 80] Brian K. Reid and Janet H. Walker.
Scribe User's Manual
Third edition, Unilogic, Ltd., May 1980.

[Rosenthal 81] David S. H. Rosenthal.
Methodology in Computer Graphics Re-examined.
Computer Graphics 15(2):152-162, July 1981.

[Rosenthal 82] David S. H. Rosenthal, James C. Michener, Gunther Pfaff, Rens Kessener, and Malcolm Sabin.
The Detailed Semantics of Graphics Input Devices.
Computer Graphics 16(3):33-38, July 1982.

[Scheifler 86] Robert W. Scheifler and Jim Gettys.
The X Window System.
ACM Transactions on Graphics 5(2):79-109, April 1986.

[Sedgewick 88] Robert Sedgewick.
Algorithms.
Addison-Wesley, Reading, MA, 1988.

[Shaw 86] Mary Shaw.
An Input-Output Model for Interactive Systems.
In *Proceedings of CHI '86: Human Factors in Computing Systems*, pages 261-273. ACM SIGCHI, 1986.

[Shaw 89] Mary Shaw.
Larger Scale Systems Require Higher-Level Abstractions.
In *Proceedings of Fifth International Workshop on Software Specification and Design*, pages 143-146. IEEE Computer Society, May 1989.
ACM SIGSOFT Software Engineering Notes, Volume 14 Number 3.

[Shneiderman 86] Ben Shneiderman.
Seven Plus or Minus Two Central Issues in Human-Computer Interaction.
In *Proceedings of CHI '86: Human Factors in Computing Systems*, pages 343-349. ACM SIGCHI, 1986.

[Tanner 83] Peter Tanner and William Buxton.
Some Issues in Future User Interface Management System Development.
In Gunther Pfaff (editor), *Seeheim Workshop on User Interface Management Systems*, pages 67-79. EUROGRAPHICS-Springer, 1983.

[Wegner 87] Peter Wegner.
Dimensions of Object-Based Language Design.
Sigplan Notices 22(12):168-182, December 1987.
Proceedings of OOPSLA '87: Conference on Object-Oriented Programming Systems, Languages, and Applications.

Table of Contents

1. Introduction **1**
1.1. Rationale 1
1.2. The Notion of a Design Space 2
1.3. Related Work 3

2. An Overview of the Design Space **5**
2.1. A Basic Structural Model 5
2.2. Functional Design Dimensions 6
2.2.1. The Most Important Functional Dimensions 9
2.3. Structural Design Dimensions 10

3. Design Rules for User Interface Systems **17**

4. Automating the Design Rules **19**

5. Summary **21**

Appendix A. The Design Space **23**
A.1. Functional Design Dimensions 23
A.1.1. External Requirements 23
A.1.1.1. Application Characteristics 23
A.1.1.2. User Needs 25
A.1.1.3. I/O Devices 25
A.1.1.4. Computer System Environment 26
A.1.2. Basic Interactive Behavior 28
A.1.3. Practical Considerations 28
A.2. Structural Design Dimensions 31
A.2.1. Division of Functions and Knowledge 31
A.2.2. Representation of Information 33
A.2.3. Control Flow, Communication, and Synchronization 34

Appendix B. The Design Rules **39**
B.1. Basic Division of Functions 39
B.1.1. Application Interface 39
B.1.2. Device Interface 41
B.2. Representation Issues 42
B.2.1. User Interface Definition 43
B.2.2. Application Semantic Information 44
B.2.3. Representation of Data Values 45
B.3. Control Flow and Synchronization 45
B.4. Matters of State 48
B.4.1. Representation of Interface State 48

B.4.2. Communication via Shared State 48
B.5. Mechanisms 49
B.5.1. Communication Mechanisms 49
B.5.2. Control Flow Mechanisms 51
References 53

List of Figures

Figure 1-1: A Simple Design Space 3
Figure 2-1: A Basic Structural Model for User Interface Software 5

Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events

David Garlan

Carnegie Mellon University, Pittsburgh PA 15213, USA,
garlan@cs.cmu.edu

Abstract. Developing a good software architecture for a complex system is a critically important step for insuring that the system will satisfy its principal objectives. Unfortunately, today descriptions of software architecture are largely based on informal "box-and-line" drawings that are often ambiguous, incomplete, inconsistent, and unanalyzable. This need not be the case. Over the past decade a number of researchers have developed formal languages and associated analysis tools for software architecture. In this paper I describe a number of the representative results from this body of work.

1 Introduction

The field of software architecture is concerned with the design and modeling of systems at a level of abstraction that reveals their gross structure and allows one to reason about key system properties, such as performance, reliability, and security. Typically architectural modeling is done by describing a system as a set of interacting components, where low-level implementation details are hidden, and relevant high-level system level properties (such as expected throughputs, latencies, and reliabilities) are exposed [29, 32].

Software architecture can be viewed as a level of design and system modeling that forms a bridge between requirements and code. By providing a high-level model of system structure it permits one to understand a system in much simpler terms than is afforded by code level structures, such as classes, variables, methods, and the like. Moreover, if characterized properly an architectural description should in principle allow one to argue that a system's design satisfies key requirements by appealing to abstract reasoning over the structure. Finally, an architecture forms a blueprint for implementations, indicating what are the principle loci of computation and data storage, the channels of communication, and the interfaces through which communication takes place.

To illustrate with a simple example, consider a simple pipelined dataflow architecture, in which streams of data are processed in linear fashion by a sequence of stream transformations, or "filters." When annotated with properties such as rates of processing, buffering capabilities of the channels, and expected input rates, one can typically reason about expected throughput and latency of

M. Bernardo and P. Inverardi (Eds.): SFM 2003, LNCS 2804, pp. 1–24, 2003.

the overall system. Additionally, the architectural structure likely mirrors the implementation structures For example, each filter might be implemented as a separate process communicating over buffered, asynchronous channels provided by the operating system.

Software architecture consequently plays a critical role in almost all aspects of the software development lifecycle.

Requirements specification: Architectural design allows one to determine what one can build, and what requirements are reasonable. Often an architectural sketch is necessary to assess product viability. For example, a preliminary architectural design might tell one whether subsecond response time is a feasible requirement on a new client-server system.

System design: Software architecture is a form of high-level system design. It typically determines the first, and most critical, system decomposition. A system without a well-conceived architecture is doomed to failure.

Implementation: As noted, an architecture is often the blueprint for low-level design and implementation. The components in an architectural description typically represent subsystems in the implementation, where the architectural interfaces correspond to the interfaces provided by an implementation.

Reuse: Most systems exhibit regular structures that represent instances of reusable idioms. For example signal processing systems are often designed as stream processing systems. Data-centric information systems are often designed as 3-tiered client-server systems. More generally, software architectures are a key component of *product lines* and *frameworks*. Those systems exploit architectural (and coding) regularities across a family of systems to make it possible to design and create new systems at low cost by specializing a general framework to create a particular product.

Maintenance: Software architectures facilitate maintenance by clarifying the system design, and enabling maintainers to understand the impact of changes. Since maintenance can account for well over half of a system's lifetime costs, and a substantial portion of maintenance is simply understanding a system in order to make a desired change, software architectures can be play a significant role in maintenance.

Run time adaptation: Increasingly systems are expected to operate continuously. Automated mechanisms for detecting and repairing system faults while a system is running will likely become essential capabilities in future systems. Software architecture can play an key role in supporting self adaptation, by providing a reflective model that can be used as a basis for automated repair.

Unfortunately, the potential uses of software architecture are thwarted by today's relatively informal approaches to architectural representation, documentation, and analysis. Architectural designs are, more often than not, simply informal "box-and-line" diagrams accompanied by prose. While these representations remain useful to practitioners [31] they suffer from their imprecision. Generally, it is not possible to use them for analysis, to determine with confidence whether some property holds of a system, whether a design is complete or consistent, whether an implementation conforms to an architectural design, or whether a proposed change violates an architectural principle.

In an effort to improve this situation many researchers have proposed formal notations and tools to set architectural design on a more solid engineering footing. Indeed, over the past decade dozens of architectural description languages (ADLs), numerous architectural evaluation methods, and many architectural analysis tools have been proposed by researchers [14, 23].

In the remainder of this paper, we outline some of the ways in which formal methods and notations can be brought to bear on software architecture. We begin with a brief introduction to software architecture. Next we consider various formal approaches to modeling and analyzing architectures. Then we briefly consider automated support, and conclude by listing some of the more interesting open research problems.

2 Software Architecture

Before characterizing ways in which we can apply formal modeling and analysis to software architecture, it is important to be clear about what we mean by the term. Definitions of software architecture abound. (The Software Engineering Institute's Web site catalogs more than 90 definitions [8].) A typical one is the following:

> The structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them [6].

Unfortunately, as with most definitions of software architecture, this one begs the questions: What structures? What is a component? What kinds of relationships are relevant? What is an externally visible property?

In practice there are a number of kinds of structural decompositions of a system [8, 18]. Each of these has a legitimate place in the design and description of a complex software system, and each has its associated uses with respect to modeling and analysis.

One of these is a code decomposition, in which the primary elements are code modules (classes, packages, etc.). Relationships between these elements typically determine code usage and functionality relationships (imports, calls, inherits-from, etc.). Typical analyses include dependency analysis, portability analysis, reuse analysis.

A second class of decomposition characterizes the run-time structures of a system. Elements in such descriptions include the principal components of a system that exist as a system is running (clients, servers, databases, etc.). Also important in such descriptions are the communication channels that determine how the components interact. Relationships between these elements determine which components can communicate with each other and how they do so. Analyses of these structures address run-time properties, such as potential for deadlocks and race conditions, reliability, performance, and security. Whether a particular analysis can be performed will usually depend on the kind of system. For example, a queueing theoretic analysis might only be valid for a system composed of components that process streams of requests submitted by clients. Or, a schedulability analysis might only be valid for a system in which each component is treated as a periodic process.

Other structural representations might emphasize the physical context in which a system is deployed (processors, networks etc.), or developed (organizational teams or business units).

In this paper we focus on the second of these classes of structure: run-time decompositions emphasizing the principal computational elements and their communication channels. Sometimes this is referred to as the "component and connector" viewtype [8]. Indeed, in what follows, unless otherwise indicated, when we refer to the software architecture a system, we will mean a component and connector architectural view of it.

While systems can in principle be described as arbitrary compositions of components and connectors, in practice there are a number of benefits to constraining the design space for architectures by associating an *architectural style* with the architecture. An architectural style typically defines a vocabulary of types for components, connectors, interfaces, and properties together with rules that govern how elements of those types may be composed.

Requiring a system to conform to a style has many benefits, including support for analysis, reuse, code generation, and system evolution [11, 34, 7]. Moreover, the notion of style often maps well to widely-used component integration infrastructures (such as EJB, HLA, CORBA), which prescribe the kinds of components allowed and the kinds of interactions that may take place between them.

3 Formal Approaches to Software Architecture

Since architectural description is a multi-faceted problem, it is helpful to classify the properties of interest into several broad categories:

Structure: What are the principal components and the connectors that allow those components to communicate? What kinds of interfaces do components provide? What are the boundaries of subsystem encapsulation? Do the structures conform to any constraints on topology? Is the design complete?

Design Constraints: What design decisions should not change over time? What assumptions are being made that should be preserved in the face of future modification, or dynamically evolving architectures?

Style: What are the constraints implied by the architectural style? Does a given system conform to constraints of a given architectural style? What analyses are appropriate for a particular architectural style. What are the relationships between different architectural styles? Is it possible to combine two styles to produce a third one?

Behavior: What is the abstract behavior of each of the components? What are the protocols of communication that are required for two components to interact? Are the components behaviorally compatible? How does a system evolve structurally over time? Can we guarantee that all possible structures that emerge at run time will satisfy some property?

Refinement: Does a more detailed representation, and in particular a concrete implementation, respect the structure and properties of an architectural design?

Let us now consider how formal representations of software architecture can address many of these questions.

3.1 Formalizing Architectural Structure

Over the past decade there has been considerable research devoted to the problem of providing more precise ways to characterize the structure of software architectures, and to derive properties of those structures. Indeed, more than a dozen Architecture Description Languages (or *ADLs*) have been proposed. These notations usually provide both a conceptual framework and a concrete syntax for modeling software architectures. They also typically provide tools for parsing, unparsing, displaying, compiling, analyzing, or simulating architectural descriptions written in their associated language.

Examples of ADLs include Aesop [11], Adage [9], C2 [22], Darwin [20], Rapide [19], SADL [26], UniCon [30], Meta-H [7], and Wright [4]. While all of these languages are concerned with architectural design, each provides certain distinctive capabilities: 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 support a mixture of heterogeneous component and connector types; Wright supports the formal specification and analysis of interactions between architectural components.

Although there is considerable diversity in the capabilities of different ADLs, all share a similar conceptual basis [23], that determines a common foundation for architectural description. The main elements are:

- *Components* represent the primary computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of software architectures. Typical examples of components include such things as clients, servers, filters, objects, blackboards, and databases. In most ADLs components may have multiple interfaces, each interface defining a point of interaction between a component and its environment.
- *Connectors* represent interactions among components. Computationally speaking, connectors mediate the communication and coordination activities among components. That is, they provide the “glue” for architectural

designs, and intuitively, they correspond to the lines in box-and-line descriptions. Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast. But connectors may also represent more complex interactions, such as a client-server protocol or a SQL link between a database and an application. Connectors also have interfaces that define the roles played by the various participants in the interaction represented by the connector.
- *Systems* represent configurations (graphs) of components and connectors. In modern ADLs a key property of system descriptions is that the overall topology of a system is defined independently from the components and connectors that make up the system. (This is in contrast to most programming language module systems where dependencies are wired into components via import clauses.) Systems may also be hierarchical: components and connectors may represent subsystems that have "internal" architectures.
- *Properties* represent semantic information about a system and its components that goes beyond structure. As noted earlier, different ADLs focus on different properties, but virtually all provide *some* way to define one or more extra-functional properties together with tools for analyzing those properties. For example, some ADLs allow one to calculate overall system throughput and latency based on performance estimates of each component and connector [33].
- *Constraints* represent claims about an architectural design that should remain true even as it evolves over time. Typical constraints include restrictions on allowable values of properties, topology, and design vocabulary. For example, an architecture might constrain its design so that the number of clients of a particular server is less than some maximum value.
- *Styles* represent families of related systems. An architectural *style* typically defines a vocabulary of design element types and rules for composing them [32]. Examples include dataflow architectures based on graphs of pipes and filters, blackboard architectures based on shared data space and a set of knowledge sources, and layered systems. Some architectural styles additionally prescribe a framework[1] as a set of structural forms that specific applications can specialize. Examples include the traditional multistage compiler framework, 3-tiered client-server systems, the OSI protocol stack, and user interface management systems.

As a very simple illustrative example, consider a simple containing a client and server component connected by a RPC connector. The server itself might be represented by a subarchitecture. Properties of the connector might include the protocol of interaction that it requires. Properties of the server might include the

[1] Terminology distinguishing different kinds of families of architectures is far from standard. Among the terms used are "product-line frameworks," "component integration standards," "kits," "architectural patterns," "styles," "idioms," and others. For the purposes of this paper, the distinctions between these kinds of architectural families is less important than the fact that they all represent a set of architectural instances.

average response time for requests. Constraints on the system might stipulate that no more than five clients can ever be connected to this server and that servers may not initiate communication with a client. The style of the system might be a "client-server" style in which the vocabulary of design includes clients, servers, and RPC connectors.

This conceptual basis of ADLs provides a natural way to model the run-time architectures of systems. First, ADLs allow one to describe compositions of components precisely, making explicit the ways in which those components communicate. Second, they support hierarchical descriptions and encapsulation of subsystems as components in a larger system. Third, they support the specification and analysis of non-functional properties. Fourth, many ADLs provide an explicit home for describing the detailed semantics of communication infrastructure (through specification of connector types). Fifth, ADLs allow one to define constraints on system composition that make clear what kinds of compositions are allowed. Finally, architectural styles allow one to make precise the differences between kinds of component integration standards.

To be concrete, we now describe a representative ADL, called Acme [13] Acme supports the definition of four distinct aspects of architecture. First is structure—the organization of a system as a set of interacting parts. Second is properties of interest—information about a system or its parts that allow one to reason abstractly about overall behavior (both functional and extra-functional). Third is constraints—guidelines for how the architecture can change over time. Fourth is types and styles—defining classes and families of architecture.

Structure Architectural structure is defined in Acme using seven core types of entities: *components, connectors, systems, ports, roles, representations, and rep-maps.* Consistent with the vocabulary outlined earlier, Acme *components* represent computational elements and data stores of a system. A component may have multiple interfaces, each of which is termed a *port.* A port identifies a point of interaction between the component and its environment, and can represent an interface as simple as a single procedure signature. Alternatively, a port can define a more complex interface, such as a collection of procedure calls that must be invoked in certain specified orders, or an event multicast interface.

Acme *connectors* represent interactions among components. Connectors also have interfaces that are defined by a set of *roles.* Each role of a connector defines a participant of the interaction represented by the connector. Binary connectors have two roles such as the *caller* and *callee* roles of an RPC connector, the *reading* and *writing* roles of a pipe, or the *sender* and *receiver* roles of a message passing connector. Other kinds of connectors may have more than two roles. For example an event broadcast connector might have a single *event-announcer* role and an arbitrary number of *event-receiver* roles.

Acme *systems* are defined as graphs in which the nodes represent components and the arcs represent connectors. This is done by identifying which component ports are *attached* to which connector roles.

Figure 1 contains an Acme description of the simple architecture described above. A *client* component is declared to have a single *send-request* port, and the server has a single *receive-request* port. The connector has two roles designated *caller* and *callee*. The topology of this system is defined by listing a set of *attachments* that bind component ports to connector roles. In this case, the client's requesting port is bound to the rpc's caller role, and the servers's request-handling port is bound to the rpc's callee role.

```
System simple_cs = {
  Component client = { Port sendRequest }
  Component server = { Port receiveRequest }
  Connector rpc  = { Roles {caller, callee} }
  Attachments : {
      client.sendRequest to rpc.caller ;
      server.receiveRequest to rpc.callee }
}
```

Fig. 1. Simple Client-Server System in Acme.

To support hierarchical descriptions of architectures, Acme permits any component or connector to be represented by one or more detailed, lower-level descriptions. Each such description is termed a *representation*.

When a component or connector has an architectural representation there must be some way to indicate the correspondence between the internal system representation and the external interface of the component or connector that is being represented. A *rep-map* (short for "representation map") defines this correspondence. In the simplest case a rep-map provides an association between internal ports and external ports (or, for connectors, internal roles, and external roles).[2] In other cases the map may be considerably more complex.

Figures 2 illustrates the use of representations in elaborating the simple client-server example. In this case, the *server* component is elaborated by a more detailed architectural representation.

Properties The seven classes of design element outlined above are sufficient for defining the *structure* of an architecture as a graph of components and connectors. However, there is more to architectural description than structure. But what exactly? Looking at the range of ADLs, each typically has its own forms of auxiliary information that determines such things as the run-time semantics of the system, protocols of interaction, scheduling constraints, and resource consumption. Clearly, the needs for documenting extra-structural properties of a system's architecture depend on the nature of the system, the kinds of analyses required, the tools at hand, and the level of detail included in the description.

[2] Note that rep-maps are not connectors: connectors define paths of interaction, while rep-maps identify an abstraction relationship between sets of interface points.

```
System simpleCS = {
  Component client = { ... }
  Component server = {
        Port receiveRequest;
        Representation serverDetails = {
          System serverDetailsSys = {

            Component connectionManager = {
               Ports { externalSocket; securityCheckIntf; dbQueryIntf } }

            Component securityManager = {
               Ports { securityAuthorization; credentialQuery; } }

            Component database = {
               Ports { securityManagementIntf; queryIntf; } }

            Connector SQLQuery = { Roles { caller; callee } }
            Connector clearanceRequest = { Roles { requestor; grantor } }
            Connector securityQuery = {
               Roles { securityManager; requestor } }
            Attachments {
               connectionManager.securityCheckIntf to clearanceRequest.requestor;
               securityManager.securityAuthorization to clearanceRequest.grantor;
               connectionManager.dbQueryIntf to SQLQuery.caller;
               database.queryIntf to SQLQuery.callee;
               securityManager.credentialQuery to securityQuery.securityManager;
               database.securityManagementIntf to securityQuery.requestor; }

       }
       Bindings { connectionManager.externalSocket to server.receiveRequest }
    }
  }
  Connector rpc  = { ... }
  Attachments { client.send-request to rpc.caller ;
                server.receive-request to rpc.callee }
```

Fig. 2. Client-Server System with Representation.

To accommodate the open-ended requirements for specification of auxiliary information, Acme supports annotation of architectural structure with arbitrary lists of properties. Figure 3 shows the simple client-server system elaborated with several properties. In the figure, properties document such things as the client's expected request rate and the location of its source code. For the *rpc* connector, properties document the protocol of interaction described as a Wright specification [4] (described in Section 3.4).

Properties serve to document details of an architecture relevant to its design and analysis. However, from Acme's point of view properties are uninterpreted values—that is, they have no intrinsic semantics. Properties become useful, however, when tools use them for analysis, translation, display, and manipulation.

```
System simple_cs = {
  Component client = {
        Port sendRequest;
        Properties { requestRate : float = 17.0;
                     sourceCode : externalFile = "CODE-LIB/client.c" }}

  Component server = {
        Port receiveRequest;
        Properties { idempotent : boolean = true;
                     maxConcurrentClients : integer = 1;
                     multithreaded : boolean = false;
                     sourceCode : externalFile = "CODE-LIB/server.c" }}

  Connector rpc  = {
        Role caller;
        Role callee;
        Properties { synchronous : boolean = true;
                     maxRoles : integer = 2;
                     protocol : WrightSpec = "..." }}

  Attachments {
     client.send-request to rpc.caller ;
     server.receive-request to rpc.callee }
}
```

Fig. 3. Client-Server System with Properties.

3.2 Formalizing Architectural Design Constraints

One of the key ingredients of an architecture model is a set of design constraints that determine how an architectural design is permitted to evolve over time. Acme uses a constraint language based on first order predicate logic. That is, design constraints are expressed as predicates over architectural specifications. The constraint language includes the standard set of logical constructs (conjunction, disjunction, implication, quantification, and others). It also includes a number of special functions that refer to architecture-specific aspects of a system. For example, there are predicates to determine if two components are connected, and if a component has a particular property. Other functions return the set of components in a given system, the set of ports of a given component, the set of representations of a connector, and so forth. Figure 4 lists a representative set of example functions. (For a detailed description see [25].)

Constraints can be associated with any design element of an architectural model. The scope of the constraint is determined by that association. For example, if a constraint is attached to a system then it can refer to any of the design elements contained within it (components, connectors, and their parts). On the other hand, a constraint attached to a component can only refer to that compo-

Connected(comp1, comp2)	True if component comp1 is connected to component comp2 by at least one connector
Reachable(comp1, comp2)	True if component comp2 is in the transitive closure of Connected(comp1, *)
HasProperty(elt, propName)	True if element elt has a property called propName
HasType(elt, typeName)	True if element elt has type typeName
SystemName.Connectors	The set of connectors in system SystemName
ConnectorName.Roles	The set of the roles in connector ConnectorName

Fig. 4. Sample Functions for Constraint Expressions.

nent (using the special keyword *self*, and its parts (that is, its ports, properties, and representations).

To give a few examples, consider the following constraints that might be associated with a system:

```
connected(client, server)
```

will be true if the components named *client* and *server* are connected directly by a connector.

```
Forall conn : connector in systemInstance.Connectors @ size(conn.roles)
= 2
```

will be true of a system in which all of the connectors are binary connectors.

```
Forall conn : connector in systemInstance.Connectors @
    Forall r : role in conn.Roles @
        Exists comp : component in systemInstance.Components @
            Exists p : port in comp.Ports @ attached(p,r) and (p.protocol
= r.protocol)
```

will be true when all connectors in the system are attached to a port, and the attached (port, role) pair share the same protocol. Here the port and role protocol values are represented as properties of the port and role design elements.

Constraints can also define the range of legal property values, as in

```
self.throughputRate >= 3095
```

and indicate relationships between properties, as in

```
comp.totalLatency =
    (comp.readLatency + comp.processingLatency + comp.writeLatency)
```

Constraints may be attached to design elements in one of two ways: as an *invariant* or a *heuristic*. In the first case, the constraint is taken to be a rule that cannot be violated. In the second case, the constraint is taken to be a rule that should be observed, but may be selectively violated. Tools that check for consistency will naturally treat these differently. A violation of an invariant makes the architectural specification invalid, while a violation of a heuristic is treated as a warning.

Figure 5 illustrates how constraints might be used for a hypothetical *MessagePath* connector. In this example an invariant prescribes the range of legal buffer sizes, while a heuristic prescribes a maximum value for the expected throughput.

```
System messagePathSystem = {
  ...
  Connector MessagePath = {
      Roles {source; sink;}
      Property expectedThroughput : float =  512;
      Invariant (queueBufferSize >= 512) and (queueBufferSize <= 4096);
      Heuristic expectedThroughput <= (queueBufferSize / 2);
   }
}
```

Fig. 5. *MessagePath* Connector with Invariants and Heuristics.

3.3 Formalizing Architectural Style

An important general capability for the description of architectures is the ability to define styles—or families—of systems. Styles allow one to define a domain-specific or application-specific design vocabulary, together with constraints on how that vocabulary can be used. This in turn supports packaging of domain-specific design expertise, use of special-purpose analysis and code-generation tools, simplification of the design process, and the ability to check for conformance to architectural standards.

The basic building block for defining styles in Acme is a type system that can be used to encapsulate recurring structures and relationships. Using Acme one can define types of components, connectors, ports, and roles. Each such type provides a type name and a list of required substructure, properties, and constraints.

Figure 6 illustrates the definition of a *Client* component type. The type definition specifies that any component that is an instance of type *Client* must have at least one port called *Request* and a property called *request-rate* of type float. Further, the invariants associated with the type require that all ports of a *Client* component have a *protocol* property whose value is *rpc-client*, that no client more than 5 ports, that a component's request rate is larger greater than 0. Finally, there is a heuristic indicating that the request-rate should be less than 100.

```
Component Type Client = {
    Port Request = {Property protocol: CSPprotocolT};
    Property request-rate: Float;
    Invariant Forall p in self.Ports @ p.protocol = rpc-client;
    Invariant size(self.Ports) <= 5;
    Invariant request-rate >= 0;
    Heuristic request-rate < 100;
}
```

Fig. 6. Component Type "Client."

An Acme style, or *family*[3] is defined by specifying a set of types and a set of constraints. The types provide the design vocabulary for the style. The constraints determine how instances of those types can be used.

Figure 7 illustrates the definition of a "Pipe and Filter" style, together with a sample system declaration using the style. The style defines two component types, one connector type, and one property type. The single invariant of this family prescribes that all connectors must be pipes. The system *simplePF* is then defined as an instance of the style. This declaration allows the system to make use of any of the types in the style, and it must satisfy all of the style's invariants.

But what does it mean for an instance to satisfy a type? In Acme, types are interpreted as predicates, and asserting that an instance satisfies a type is the same as asserting that it satisfies the predicate denoted by the type. The predicate associated with a type is constructed by viewing declared structure as asserting the *existence* of that structure in each instance. In other words, a type defines the *minimal* structure of its instances.[4] (Hence, in the example of Figure 7 it is essential to include the invariant asserting that all connectors have type *pipe*.)

The use of a predicate-based type system has several important consequences. First, design elements (and systems) can have an arbitrary number of types. For example, the fact that a structural element is declared to be of a particular type, does not preclude it from satisfying other type specifications. This is an important property since it permits, for example, a system to be considered a valid instance of a style, even though it was not explicitly declared as such.

Second, the use of invariants fits smoothly within the type system. Adding a invariant to a structural type or family simply conjoins that predicate with the others in the type. This means that the type system becomes quite expressive – essentially harnessing predicate logic to create useful type distinctions.

[3] For historical reasons a "style" in Acme is termed a "family."

[4] The semantics of the Acme type system is similar to – but considerably simpler than – that of other predicate-based type systems, such as the one used by PVS [28]. For a formal treatment of the semantics, see [25].

```
Family PipeFilterFam = {

  Component Type FilterT = {
        Ports { stdin; stdout; };
        Property throughput : int;
  };
  Component Type UnixFilterT extends FilterT with {
        Port stderr;
        Property implementationFile : String;
  };
  Connector Type PipeT = {
        Roles { source; sink; };
        Property bufferSize : int;
  };
  Property Type StringMsgFormatT = Record [ size:int; msg:String; ];
  Invariant Forall c in self.Connectors @ HasType(c, PipeT);

}

System simplePF : PipeFilterFam = {

    Component smooth : FilterT = new FilterT
    Component detectErrors : FilterT;
    Component showTracks : UnixFilterT = new UnixFilterT extended with {
        Property implementationFile : String = "IMPL_HOME/showTracks.c";
    };

    // Declare the system's connectors
    Connector firstPipe : PipeT;
    Connector secondPipe : PipeT;

    // Define the system's topology
    Attachments { smooth.stdout to firstPipe.source;
                  detectErrors.stdin to firstPipe.sink;
                  detectErrors.stdout to secondPipe.source;
                  showTracks.stdin to secondPipe.sink; }
}
```

Fig. 7. Definition of a Pipe-Filter Family.

Third, the process of type checking becomes one of checking satisfaction of a set of predicates over declared structures. Hence, types play two useful roles: (a) they encapsulate common, reusable structures and properties, and (b) they support a powerful form of checkable redundancy.

The use of predicates does, however, raise the issue that, in general, checking for satisfaction of predicates is not decidable. Therefore, systems that rely on predicate-based type systems usually do so with the aid of a theorem prover (for example, PVS [28]). In Acme, however, we constrain the expressiveness of types so that type checking remains decidable by ensuring that quantification is only over finite sets of elements. (Finiteness comes from the fact that Acme structures can only declare a finite number of subparts – components, ports, representations, and others.)

3.4 Formalizing Architectural Behavior

In addition to formal modeling of architectural structure, properties, constraints and styles, it is also useful to be able to model and analyze architectural behavior. By associating behavior with architectures, we are able to express much richer semantic models, capturing things such as the fact that a pipe provides buffered, order-preserving data transmission, or that a given component will call the services of another component in some particular order. This in turn allows us to attach analyze important properties, such as system deadlocks, race conditions, and interface incompatibilities.

In principle there are many possible ways one might specify behavior of the elements in an architectural model. Indeed, almost any formalism can be used, and researchers have experimented with formal techniques ranging from pre-post conditions [1], process algebras [4, 20], statecharts [5], POSets [19], rewrite rules [17], and many others.

However, all of these have a similar flavor: (1) they document the individual elements with behavior characterized in terms of abstract events, states and transitions, and (2) they then perform various composition checks or simulations to test for aggregate behavior, mismatches, deadlocks, and other anomalies.

Wright. To illustrate how this can be done, consider the Wright architecture specification language [4]. Wright adopts an approach based on the process algebra CSP [16]. Specifically it associates a CSP-like process with each component, each component interface (port), each connector, and each connector interface (role). The overall behavior is then a set of interacting protocols.

The notation used is a subset of CSP, containing the following elements:

- **Processes and Events:** A process describes an entity that can engage in communication events.[5] Events may be primitive or they can have associated data (as in `e?x` and `e!x`, representing input and output of data, respectively). The simplest process, STOP, is one that engages in no events. The event $\surd$ is used represent the "success" event. The set of events that a process, P, understands is termed the "alphabet of P," or αP.
- **Prefixing:** A process that engages in event `e` and then becomes process `P` is denoted $e{\rightarrow}P$.

[5] It should be clear that by using the term "process" we do not mean that the implementation of the protocol would actually be carried out by a separate operating system process. That is to say, processes are logical entities used to specify the components and connectors of a software architecture.

- **Alternative:** ("deterministic choice") A process that can behave like P or Q, where the choice is made by the environment, is denoted $P \,\Box\, Q$. ("Environment" refers to the other processes that interact with the process.)
- **Decision:** ("non-deterministic choice") A process that can behave like P or Q, where the choice is made (non-deterministically) by the process itself, is denoted $P \sqcap Q$.
- **Named Processes:** Process names can be associated with a (possibly recursive) process expression. Unlike CSP, however, we restrict the syntax so that only a finite number of process names can be introduced. We do not permit, for example, names of the form $Name_i$, where i can range over the positive numbers.

In process expressions → associates to the right and binds tighter than either $\Box$ or $\sqcap$. So $e{\rightarrow}f{\rightarrow}P \,\Box\, g{\rightarrow}Q$ is equivalent to $(e{\rightarrow}(f{\rightarrow}P)) \,\Box\, (g{\rightarrow}Q)$.

In addition to this standard notation from CSP we introduce three notational conventions. First, we use the symbol § to represent a successfully terminating process. This is the process that engages in the success event, $\surd$, and then stops. (In CSP, this process is called SKIP.) Formally, $\S \overset{\mathrm{def}}{=} \surd{\rightarrow}\mathrm{STOP}$. Second, we allow the introduction of scoped process names, as follows: **let** $Q = expr1$ **in** R. Third, as in CSP, we allow events and processes to be labeled. The event e labeled with l is denoted $l.e$. The operator ":" allows us to label all of the events in a process, so that $l : P$ is the same process as P, but with each of its events labeled. For our purposes we use the variant of this operator that does not label $\surd$. We use the symbol Σ to represent the set of all unlabeled events.

This subset of CSP defines processes that are essentially finite state. It provides sequencing, alternation, and repetition, together with deterministic and non-deterministic event transitions.

Connector Description. To see how this is used let us consider first how a connector is specified. A connector type is specified by a set of *roles* processes and a *glue* process. The roles describe the expected local behavior of each of the interacting parties. For example, the client-server connector illustrated earlier would have a client role and a server role. The client role process might describe the client's behavior as a sequence of alternating requests for service and receipts of the results. The server role might describe the server's behavior as the alternate handling of requests and return of results. The glue specification describes how the activities of the client and server roles are coordinated. It would say that the activities must be sequenced in the order: client requests service, server handles request, server provides result, client gets result.

This is how it would be written using the notation just outlined.

```
connector Service =
    role Client = request!x→ result?y → Client ⊓ §
    role Server = invoke?x→ return!y → Server □ §
    glue = Client.request?x→ Service.invoke!x
                →Service.return?y→Client.result!y→glue
                □ §
```

The Server role describes the communication behavior of the server. It is defined as a process that repeatedly accepts an invocation and then returns; or it can terminate with success instead of being invoked. Because we use the alternative operator ($[\,]$), the choice of invoke or $\surd$ is determined by the environment of that role (which, as we will see, consists of the other roles and the glue).

The Client role describes the communication behavior of the user of the service. Similar to Server, it is a process that can call the service and then receive the result repeatedly, or terminate. However, because we use the decision operator ($\sqcap$) in this case, the choice of whether to call the service or to terminate is determined by the role process itself. Comparing the two roles, note that the two choice operators allow us to distinguish formally between situations in which a given role is *obliged* to provide some services – the case of Server – and the situation where it may take advantage of some services if it chooses to do so – the case of Client.

The **glue** process coordinates the behavior of the two roles by indicating how the events of the roles work together. Here **glue** allows the Client role to decide whether to call or terminate and then sequences the remaining three events and their data.

The example above illustrates that the connector description language is capable of expressing the traditional notion of providing and using a set of services – the kind of connection supported by import/export clauses of module interconnection.

As another illustration, consider two examples of a shared data connector.

```
connector Shared Data1 =
    role User1 = set→User1 ⊓ get→User1 ⊓ §
    role User2 = set→User2 ⊓ get→User2 ⊓ §
    glue = User1.set→glue [] User2.set→glue
            [] User1.get→glue [] User2.get→glue [] §

connector Shared Data2 =
    role Initializer =
       let A = set→A ⊓ get→A ⊓ §
       in set→A
    role User = set→User ⊓ get→User ⊓ §
    glue = let Continue = Initializer.set→Continue
                          [] User.set→Continue
                          [] Initializer.get→Continue
                          [] User.get→Continue [] §
            in Initializer.set→Continue [] §
```

The first, Shared Data$_1$, indicates that the data does not require an explicit initialization value. The second, Shared Data$_2$, indicates that there is a distinguished role Initializer that must supply the initial value.

To take a more complex example, consider the following specification of a pipe connector.

```
connector Pipe =
   role Writer = write→Writer ⊓ close→§
   role Reader =
      let ExitOnly = close→§
      in let DoRead = (read→Reader
                        [] read-eof→ExitOnly)
      in DoRead ⊓ ExitOnly
   glue = let ReadOnly = Reader.read→ReadOnly
                          [] Reader.read-eof
                             →Reader.close →§
                          [] Reader.close→§
            in let WriteOnly = Writer.write→WriteOnly
                                [] Writer.close→§
            in Writer.write→glue
              [] Reader.read→glue
              [] Writer.close→ReadOnly
              [] Reader.close→WriteOnly
```

It might appear to be a simple matter to define a pipe: both the writer and the reader decide when and how many times they will write or read, after which they will each close their side of the pipe. In fact, the writer role is just that simple. The reader, on the other hand, must take other considerations into account. There must be a way to inform the reader that there will be no more data.

Connector Semantics. The intuition behind a connector description is that the roles are treated as independent processes, constrained only by the glue, which serves to coordinate and interleave the events. To make this idea precise we use the CSP parallel composition operator, $\|$, for interacting processes. The process $P_1 \| P_2$ is one whose behavior is permitted by both P_1 and P_2. That is, for the events in the intersection of the processes' alphabets, both processes must agree to engage in the event. We can then take the meaning of a connector description to be the parallel interaction of the glue and the roles, where the alphabets of the roles and glue are arranged so that the desired coordination occurs.

Hence, the *meaning of a connector description* with roles R_1, R_2, …, R_n, and glue *Glue* is the process:

$$Glue \parallel (\mathrm{R}_1\mathbf{:}R_1 \parallel \mathrm{R}_2\mathbf{:}R_2 \parallel \ldots \parallel \mathrm{R}_n\mathbf{:}R_n)$$

where R_i is the (distinct) name of role R_i, and

$$\alpha Glue = \mathrm{R}_1\mathbf{:}\Sigma \cup \mathrm{R}_2\mathbf{:}\Sigma \cup \ldots \cup \mathrm{R}_n\mathbf{:}\Sigma \cup \{\surd\}.$$

In this definition we arrange for the glue's alphabet to be the union of all possible events labeled by the respective role names (*e.g.* Client, Server), together with the $\surd$ event. This allows the glue to interact with each role. In contrast, (except for $\surd$) the role alphabets are disjoint and so each role can only interact with the glue. Because $\surd$ is not relabeled, all of the roles and glue can (and must) agree on $\surd$ for it to occur. In this way we ensure that successful termination of a connector becomes the joint responsibility of all the parties involved.

Describing Components. Thus far we have concerned ourselves with the definition of connector types. To complete the picture we must also describe the ports of components and how those ports are attached to specific connector roles in a complete software architecture.

In Wright, component ports are also specified by processes: The port process defines the expected behavior of the component at that particular point of interaction. For example, a component that uses a shared data item only for reading might be partially specified as follows:

component DataUser =
  port DataRead = get→DataRead ⊓ §
  other ports...

Since the port protocols define the actual behavior of the components when those ports are associated with the roles, the port protocol takes the place of the role protocol in the actual system. Thus, an attached connector is defined by the protocol that results from the replacement of the role processes with the associated port processes. More formally, the meaning of attaching ports $P_1 \ldots P_n$ as roles $R_1 \ldots R_n$ of a connector with glue *Glue* is the process:

$$\mathit{Glue} \parallel (\mathrm{R}_1\mathbf{:}P_1 \parallel \mathrm{R}_2\mathbf{:}P_2 \parallel \ldots \parallel \mathrm{R}_n\mathbf{:}P_n).$$

Note that this definition of attachment implies that port protocols need not be identical to the role protocols that they replace. This is advantageous because it allows greater opportunities for reuse. For instance, in the above example, the DataUser component should be able to interact with another component (via a shared data connector) even though it never needs to set. As another example, we would expect to be able to attach a File port as the Reader role of a pipe (as is commonly done in Unix when directing the output of a pipe to a file).

But this raises an important question: when is a port "compatible" with a role? For example, it would be reasonable to forbid DataRead to be used as the Initializer role for the Shared Data$_2$ connectors, since it requires an initial set; clearly DataRead will never provide this event.

Analyzing Architectural Behavior. Once one has a formal definition of behavior there are a number of analyses that one can perform. The most obvious one is checking that a connector is well-formed. That is to say, that the Glue in combination with the roles does not lead to deadlock. Another useful check is to investigate race conditions. This can be done by checking whether certain events can ever occur out of order.

Yet another check is to answer questions like "what ports may be used in this role?" At first glance it might seem that the answer is obvious: simply check that the port and role protocols are equivalent. But as illustrated earlier, it is important to be able to attach a port that is not identical to the role. On the other hand, we would like to make sure that the port fulfills its obligations to the interaction. For example, if a role requires an initialization as the first operation (*cf.*, the shared data example), we would like to guarantee that any port actually performs it.

Informally, we would like to be able to guarantee that an attached port process always acts in a way that the corresponding role process is capable of acting. This can be recast as follows: When in a situation predicted by the protocol, the port must always continue the protocol in a way that the role could have.

In CSP this intuitive notion is captured by the concept of refinement. Roughly, process P_2 refines P_1 (written $P_1 \sqsubseteq P_2$) if the behaviors of P_1 include those of P_2. Technically, the definition is given in terms of the failures/divergences model of CSP [16, Chapter 3]. For various technical reasons, however, the actual definition of compatibility is a little more complex to define, although it captures the same essential idea of refinement. (See [4] for details.)

As another check, one can investigate whether a port can be left unattached. This can be done by seeing if the port will deadlock when connected to a "do nothing" connector. Other checks are described in detail in [2].

Analyzing Reconfigurable Architectures Thus far the analysis has assumed a *static* architecture: that is, the structure of the architecture does not change during the execution of a system. While this is often a useful approximation to systems, clearly in the general case systems do evolve structurally. At the very least, during initialization the system must be created, and this is not likely to be an atomic operation.

As another example, consider a simple client-server system, such as the one illustrated earlier, but that allows for the possibility that a server may crash. In such cases the system might reconfigure itself so that the client uses a backup server. This can be done by adding a new connector during run time. One of the things we would like to guarantee for such a system is that no client requests are lost. This requires some constraints on when reconfiguration can happen.

Some work has been done to address these issues, although comparatively that work is relatively sparse. In our own work we showed how to extend Wright to handle dynamically changing topologies [3]. Others have looked at ways to use the Pi Calculus to specify such things [20]. Others have looked at graph grammars [24] and category-theoretic approaches [35]. Unfortunately, in all of these cases the complexity of the specification becomes drastically higher, and the models become much less tractable for static analysis.

4 Automated Support

For all of the formal approaches outlined earlier, researchers have developed numerous tools to aid in the modeling and analysis process for architects. Broadly speaking there are three general categories of tools:

1. **Design Assistants:** These tools tend to focus on providing a graphical front end to allow architects to develop designs. Typically they provide a pallet of component and connector types that can be instantiated to create system descriptions. Typical examples are environments such as C2 [22], MetaH [7], Aesop [11], and Darwin [20].
2. **Design Checkers:** While automated support for architectural creation and browsing is valuable, to be effective one must also provide analysis capabilities. Hence, a number of tools have been created to perform various checks. For example AcmeStudio [25] checks for violations of design constraints. Wright provides a tool for performing the checks outlined earlier. Those checks are based on the use of the FDR [10] model checker for CSP. Kramer and Magee demonstrate how to use their LTSA tool to check specifications written in their process algebra, FSP [21].
3. **Code Generators:** In many cases a formal definition of an architecture can be used to generate system code. For example, the UniCon system handles the generation of connector code for a wide variety of connector types [30]. Similarly C2 can generate partial implementations in using various infrastructures to handle component interaction.

5 Conclusion and Future Prospects

As we have tried to illustrate, software architecture is a field in which formal modeling and analysis can have a major impact. While the state of practice continues to rely on informal and semi-formal descriptions, considerable research has been done to develop good formal models and associated tools for analyzing them.

But the story is far from complete and there a number of areas in which further research is needed. Here are a few.

- Scalability: Although some large case studies have been carried out (e.g., [5]), there are relatively few demonstrated success stories for large, complex industrial systems. When systems have thousands of components, it is not clear how well the representation techniques (particularly graphical ones) scale. Nor is it clear whether analyses remain tractable. For example, many analysis tools are based on model checkers, which have significant limitation on the size of the model that can be checked.
- Dynamism: As noted earlier a key issue is modeling systems whose structure changes at run time.
- Code conformance: One of the big problems is guaranteeing that an implementation conforms to its architectural specification. In situations where a

code generator is used it is often possible to guarantee conformance by construction. But more generally, given an architecture and body of code, there has been very little work on finding ways to make sure they are consistent. The main problem is that architectures (as we have discussed them) represent run-time models, whereas code is obviously a design-time artifact. In general it is undecidable whether a given body of code will generate a given architecture.

There are also some intriguing new directions being explored in the area of self-adaptive systems. Increasingly systems are required to run continuously. Moreover they must often do this in the context of environments whose resources are constantly changing (e.g., wireless bandwidth), or whose components may be changing dynamically (e.g., web services). One approach that is being investigated by a number of researchers is the incorporation of self-adaptation or self-healing into a system. The interesting question is how should one do this?

One approach is to use architectural models as the basis for system monitoring and repair [12, 15, 27]. The idea is that the architectural model becomes available at run-time in order to understand whether a system is performing optimally, and if not it can be used model to reason about reasonable repair strategies at a high level of abstraction. While work is just beginning in this area, it appears to be a promising avenue for future research.

References

[1] Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, October 1995.

[2] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[3] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998.

[4] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

[5] Robert Allen, David Garlan, and James Ivers. Formal modeling and analysis of the HLA component integration standard. In *Proceedings of of the 6th International Symposium on the Foundations of Software Engineering (FSE-6)*, Lake Buena Vista, Florida, November 1998. ACM Press.

[6] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998. ISBN 0-201-19930-0.

[7] Pam Binns and Steve Vestal. Formal real-time architecture specification and analysis. In *Tenth IEEE Workshop on Real-Time Operating Systems and Software*, New York, NY, May 1993.

[8] Paul Clements, Felix Bachmann, Len Bass, David GArlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison Wesley, 2002.

[9] L. Coglianese and R. Szymanski. DSSA-ADAGE: An Environment for Architecture-based Avionics Development. In *Proceedings of AGARD'93*, May 1993.

[10] *Failures Divergence Refinement: User Manual and Tutorial.* Formal Systems (Europe) Ltd., Oxford, England, 1.2β edition, October 1992.

[11] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 179–185. ACM Press, December 1994.

[12] David Garlan, Shang-Wen Cheng, and Bradley Schmerl. Increasing system dependability through architecture-based self-repair. In A. Romanovsky R. de Lemos, C. Gacek, editor, *Architecting Dependable Systems.* Springer-Verlag, 2003.

[13] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, page 47. Cambridge University Press, 2000.

[14] David Garlan and Dewayne Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[15] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02)*, 2002.

[16] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[17] Paola Inverardi and Alex Wolf. Formal specification and analysis of software architectures using the chemical, abstract machine model. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):373–386, April 1995.

[18] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, pages 42–50, November 1995.

[19] David C Luckham, Lary M. Augustin, John J. Kenney, James Veera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):336–355, April 1995.

[20] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference, ESEC'95*, September 1995.

[21] Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs.* Wiley, 1999.

[22] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *SIGSOFT'96: Proceedings of the Fourth ACM Symposium on the Foundations of Software Engineering.* ACM Press, October 1996.

[23] Nenad Medvidovic and Richard N. Taylor. Architecture description languages. In *Software Engineering – ESEC/FSE'97*, volume 1301 of *Lecture Notes in Computer Science*, Zurich, Switzerland, September 1997. Springer.

[24] Daniel Le Metayer. Software architecture styles as graph grammars. In *Proceedings of the Fourth ACM Symposium on the Foundations of Software Engineering.* ACM SIGSOFT, October 1996.

[25] Robert T. Monroe. *Rapid Develpomentof Custom Software Design Environments.* PhD thesis, Carnegie Mellon University, July 1999.

[26] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):356–372, April 1995.

[27] P. Oriezy et al. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[28] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.

[29] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[30] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):314–335, April 1995.

[31] Mary Shaw and David Garlan. Formulations and formalisms in software architecture. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, Lecture Notes in Computer Science, Volume 1000. Springer-Verlag, 1995.

[32] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[33] Bridget Spitznagel and David Garlan. Architecture-based performance analysis. In *Tenth International Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, San Francisco, CA, June 1998.

[34] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, Jr. E. James Whitehead, Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, 22(6):390–406, June 1996.

[35] Michel Wermelinger. Formal specification and analysis of dynamic reconfiguration of software architecture. In *Proceedings of the 20th International Conference on Software Engineering*, volume 2, pages 178–179. IEEE Computer Society Press, 1998.

Software Architecture
an informal introduction

David Schmidt

Kansas State University

`www.cis.ksu.edu/~schmidt`

Outline

1. **Components and connectors**
2. **Software architectures**
3. Architectural analysis and views
4. **Architectural description languages**
5. **Domain-specific design**
6. **Product lines**
7. Middleware
8. Model-driven architecture
9. Aspect-oriented programming
10. **Closing remarks**

1. Components and connectors

Motivation for software architecture

We use already architectural idioms for describing the structure of complex software systems:

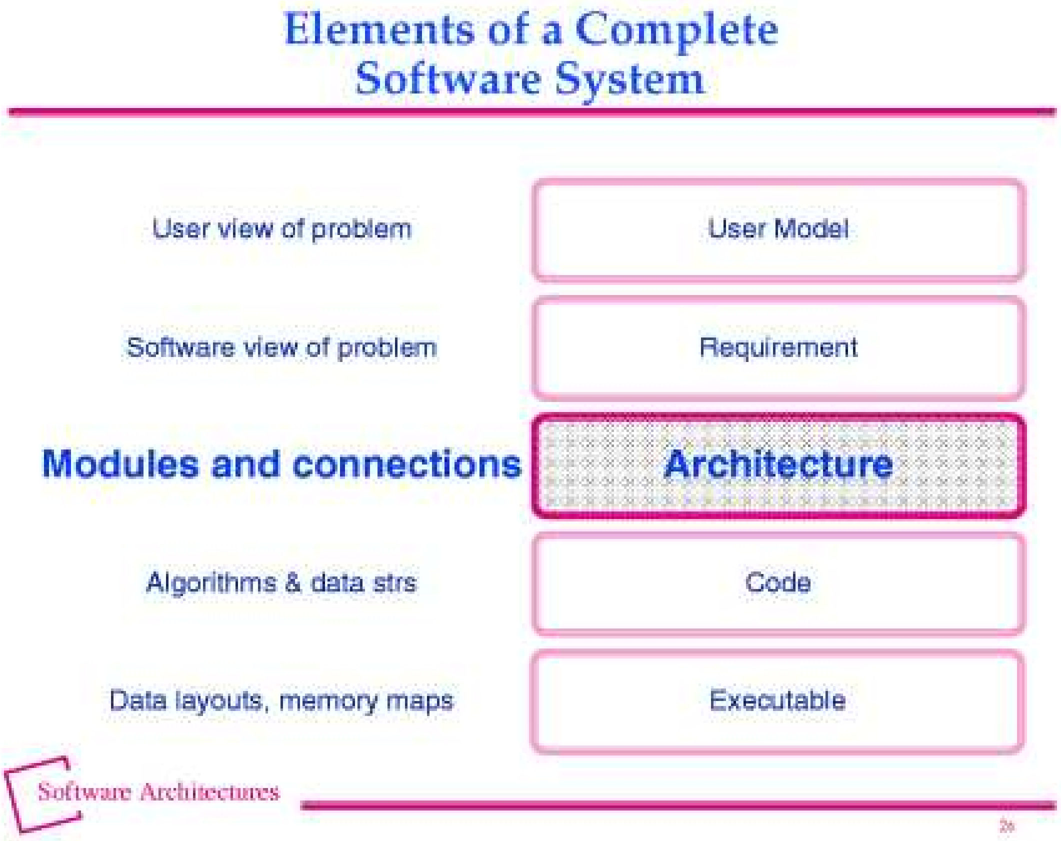
- “Camelot is based on the *client-server model* and uses remote procedure calls both locally and remotely to provide communication among applications and servers.” [Spector87]
- “The easiest way to make the canonical sequential compiler into a concurrent compiler is to *pipeline* the execution of the compiler phases over a number of processors.” [Seshadri88]
- “The ARC network follows the *general network architecture* specified by the ISO in the Open Systems Interconnection Reference Model.” [Paulk85]

Reference: David Garlan, *Architectures for Software Systems*, CMU, Spring 1998.

`http://www.cs.cmu.edu/afs/cs/project/tinker-arch/www/html/index.html`

Architectural description has a natural position in system design and implementation

A slide from one of David Garlan’s lectures:



Elements of a Complete Software System

User view of problem	User Model
Software view of problem	Requirement
Modules and connections	**Architecture**
Algorithms & data strs	Code
Data layouts, memory maps	Executable

Software Architectures

26

Reference: David Garlan, *Architectures for Software Systems*, CMU, Spring 1998.

`http://www.cs.cmu.edu/afs/cs/project/tinker-arch/www/html/index.html`

Hardware architecture

There are standardized descriptions of computer hardware architectures:

- *RISC* (reduced instruction set computer)
- *pipelined architectures*
- *multi-processor architectures*

These descriptions are well understood and successful because

(i) there are a relatively small number of design components

(ii) large-scale design is achieved by replication of design elements

In contrast, software systems use a huge number of design components and scale upwards, not by replication of existing structure, but by adding more distinct design components.

Reference: D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, October 1992.

Network architecture

Again, there are standardized descriptions:

- *star* networks
- *ring* networks
- *manhattan street* (grid) networks

The architectures are described in terms of *nodes* and *connections*.

There are only a few standard topologies.

In contrast, software systems use a wide variety of topologies.

Classical architecture

The architecture of a building is described by

- *multiple views*: exterior, floor plans, plumbing/wiring, ...
- *architectural styles*: romanesque, gothic, ...
- *style and engineering*: how the choice of style influences the physical design of the building
- *style and materials*: how the choice of style influences the materials used to construct (implement) the building.

These concepts also appear in software systems: there are

(i) *views*: control-flow, data-flow, modular structure, behavioral requirements, ...

(ii) *styles*: pipe-and-filter, object-oriented, procedural, ...

(iii) *engineering*: modules, filters, messages, events, ...

(iv) *materials*: control structures, data structures, ...

A crucial motivating concept: *connectors*

The emergence of networks, client-server systems, and OO-based GUI applications led to the conclusion that

components can be *connected* in various ways

Mary Shaw stressed this point:

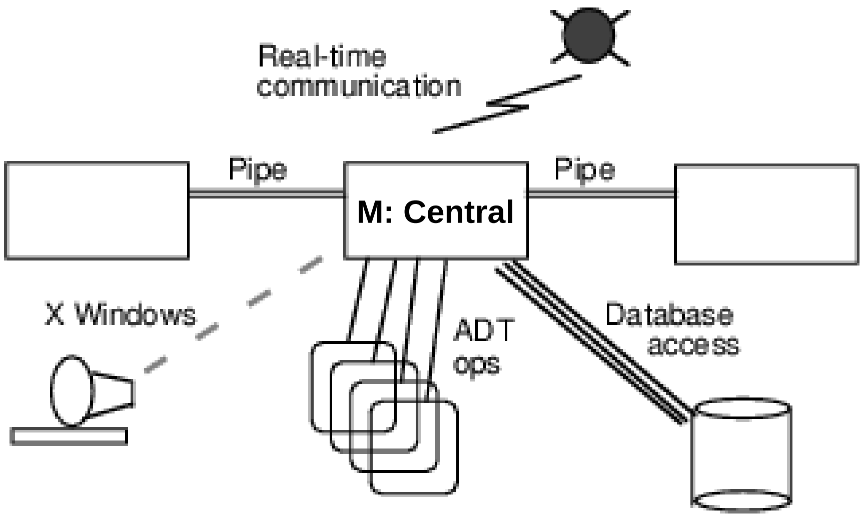


Figure 2: Revised architecture diagram with discrimination among connections

Reference: Mary Shaw, Procedure Calls are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status. Workshop on Studies of Software Design, 1993.

Shaw’s observations

Connectors are forgotten because (it appears that) there are no codes for them.

But this is because the connectors must be coded in the same language as the components, which confuses the two forms.

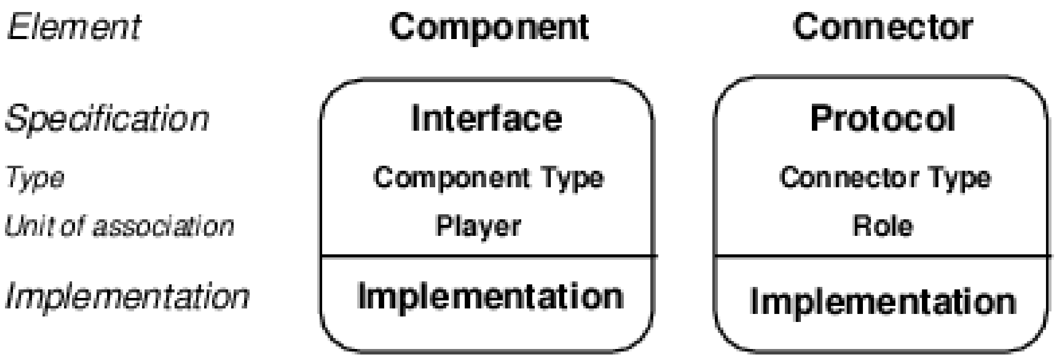
Different forms of low-level connection (synchronous, asynchronous, peer-to-peer, event broadcast) are fundamentally different yet are all represented as procedure (system) calls in programming language.

Connectors can (and should?) be coded in languages different from the languages in which components are coded (e.g., unix pipes).

Shaw’s philosophy

Components — compilation units (module, data structure, filter) — are specified by *interfaces*.

Connectors — “hookers-up” (RPC (Remote Procedure Call), event, pipe) — mediate communications between components and are specified by *protocols*.



Element	Component	Connector
Specification	Interface	Protocol
Type	Component Type	Connector Type
Unit of association	Player	Role
Implementation	Implementation	Implementation

Figure 3: Gross structure of an architecture language

Example:

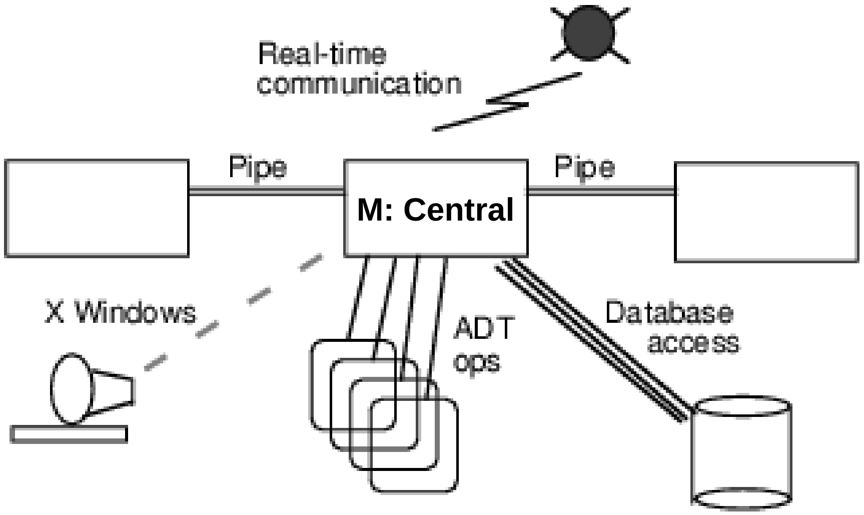


Figure 2: Revised architecture diagram with discrimination among connections

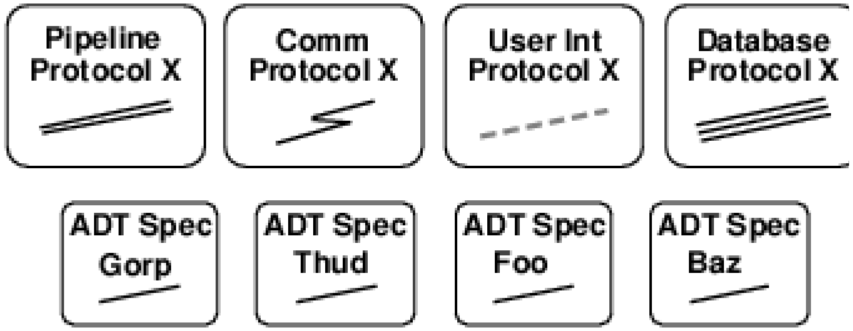
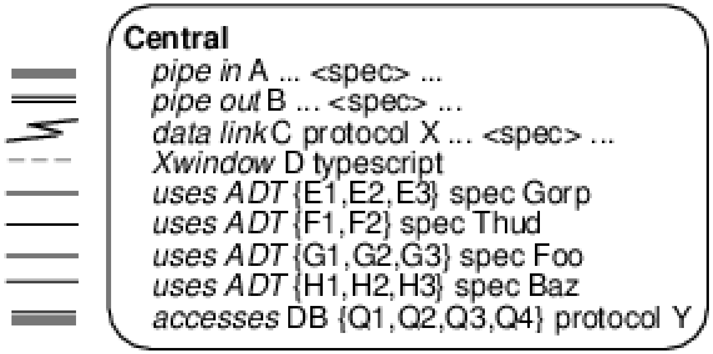


Figure 4: Constellation of protocol specifications required by exampl



Central
pipe in A ... <spec> ...
pipe out B ... <spec> ...
data link C protocol X ... <spec> ...
Xwindow D typescript
uses ADT {E1,E2,E3} spec Gorp
uses ADT {F1,F2} spec Thud
uses ADT {G1,G2,G3} spec Foo
uses ADT {H1,H2,H3} spec Baz
accesses DB {Q1,Q2,Q3,Q4} protocol Y

Figure 5: Interface specification of central component, referring to protocols

Interface **Central** is different from a Java-interface; it lists the “players” — `inA, outB, linkC, Gorp, Thud, ...` (connection points/ ports/ method invocations) — that use connectors.

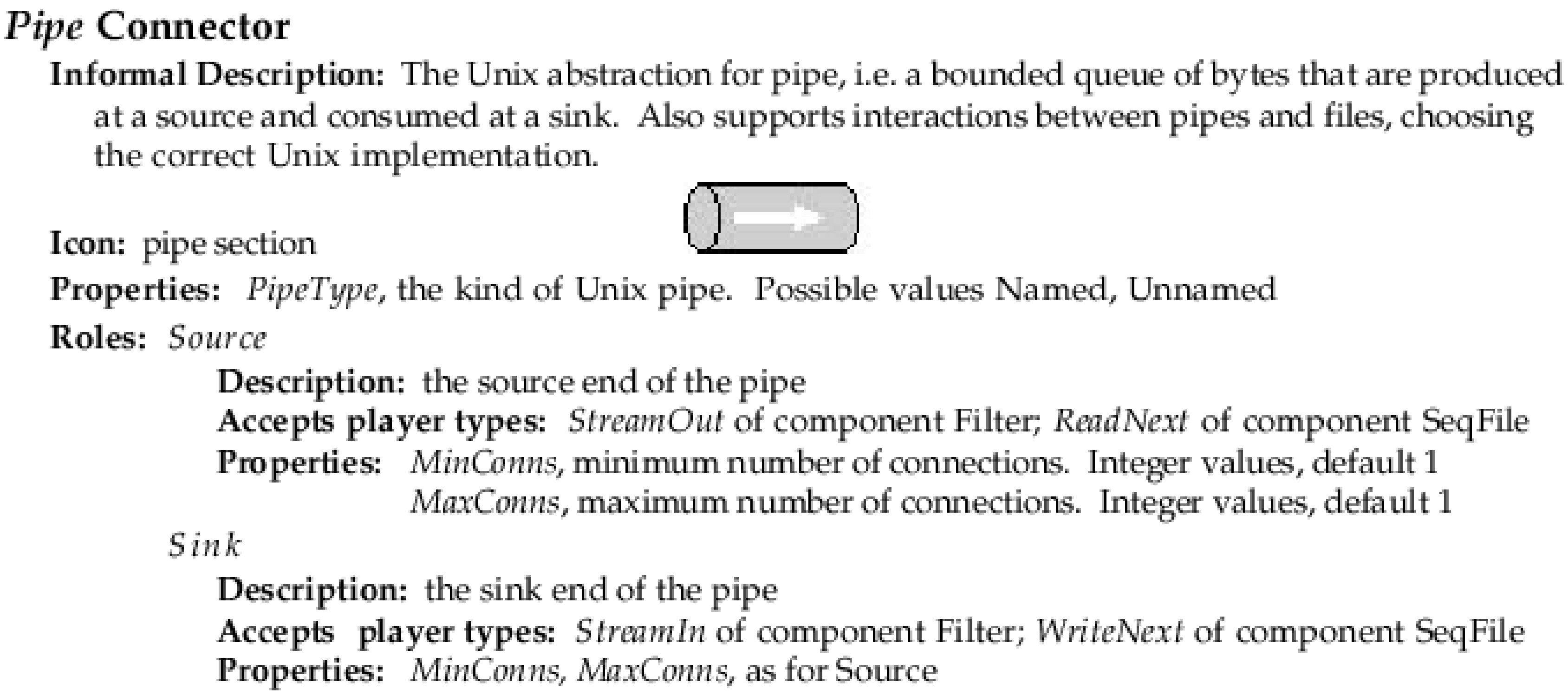
The connector's *protocol* lists

(i) the types of component interfaces it can "mediate";

(ii) orderings and invariants of component interactions;

(iii) performance guarantees.

Example: Shaw's description of a unix pipe:



Pipe **Connector**

Informal Description: The Unix abstraction for pipe, i.e. a bounded queue of bytes that are produced at a source and consumed at a sink. Also supports interactions between pipes and files, choosing the correct Unix implementation.

Icon: pipe section

Properties: *PipeType*, the kind of Unix pipe. Possible values Named, Unnamed

Roles: *Source*

Description: the source end of the pipe
Accepts player types: *StreamOut* of component Filter; *ReadNext* of component SeqFile
Properties: *MinConns*, minimum number of connections. Integer values, default 1
MaxConns, maximum number of connections. Integer values, default 1

Sink

Description: the sink end of the pipe
Accepts player types: *StreamIn* of component Filter; *WriteNext* of component SeqFile
Properties: *MinConns*, *MaxConns*, as for Source

Reference: M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and Implementations for Architectural Connections. In 3d. Int. Conf. on Configurable Distributed Systems, Annapolis, Maryland, May 1996.

Connectors can act as

- *communicators:* transfer data between components (e.g., message passing, buffering)
- *mediators:* manage shared resource access between components (e.g., reader/writer policies, monitors, critical regions)
- *coordinators:* define control flow between components (e.g., synchronization (protocols) between clients and servers, event broadcast and delivery)
- *adaptors:* connect mismatched components (e.g., a pipe connects to a file rather than to a filter)

Perhaps you have written code for a bounded buffer or a monitor or a protocol or a shared, global variable — you have written a connector!

Connectors can facilitate

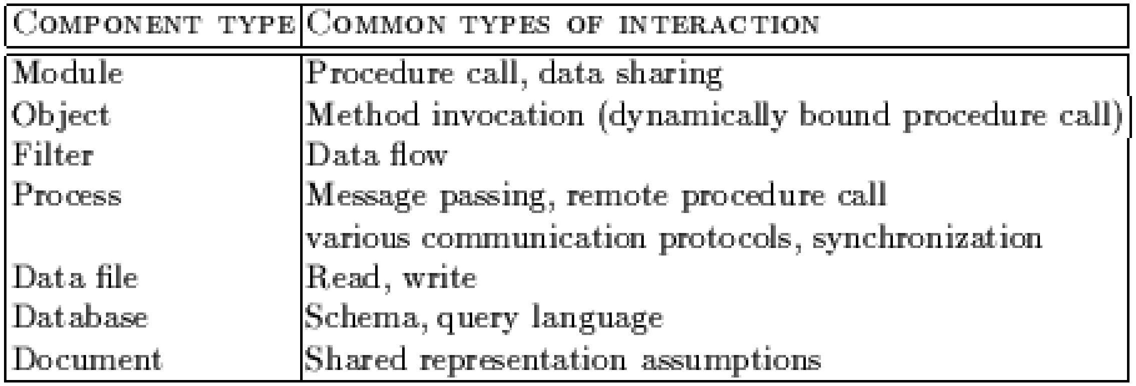
- *reuse:* components from one application are inserted into another, and they need not know about context in which they are connected

- *evolution:* components can be dynamically added and removed from connectors

- *heterogenity:* components that use different forms of communication can be connected together in the same system

A connector should have the ability to handle limited *mismatches* between connected components, via information reformatting, object-wrappers, and object-adaptors, such that the component is not rewritten — the connector does the reformatting, wrapping, adapting.

If connectors are crucial to systems building, why did we take so long to “discover” them? One answer is that components are “pre-packaged” to use certain connectors:



Component type	Common types of interaction
Module	Procedure call, data sharing
Object	Method invocation (dynamically bound procedure call)
Filter	Data flow
Process	Message passing, remote procedure call various communication protocols, synchronization
Data file	Read, write
Database	Schema, query language
Document	Shared representation assumptions

But “smart” connectors make components simpler, because the coding for interaction rests in the connectors — not the components.

The philosophy, ***system = components + connectors*** was a strong motivation for a theory of software architecture.

Reference: M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. *Computer Science Today: Recent Trends and Developments* Jan van Leeuwen, ed., Springer-Verlag LNCS, 1996, pp. 307-323.

2. Software Architecture

What is a software architecture? (Perry and Wolf)

A software architecture consists of

1. *elements*: processing elements ("functions"), connectors ("glue" — procedure calls, messages, events, shared storage cells), data elements (what "flows" between the processing elements)
2. *form*: properties (constraints on elements and system) and relationship (configuration, topology)
3. *rationale*: philosophy and pragmatics of the system: requirements, economics, reliability, performance

There can be "views" of the architecture from the perspective of the process elements, the data, or the connectors. The views might show static and dynamic structure.

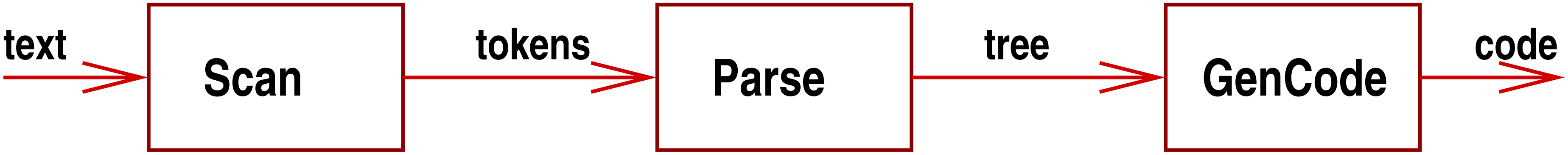
Reference: D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, October 1992.

Architectural Styles (patterns)

1. *Data-flow systems:* batch sequential, pipes and filters
2. *Call-and-return systems:* main program and subroutines, hierarchical layers, object-oriented systems
3. *Virtual machines:* interpreters, rule-based systems
4. *Independent components:* communicating systems, event systems, distributed systems
5. *Repositories (data-centered systems):* databases, blackboards
6. and there are many others, including *hybrid* architectures

The *italicized* terms are the styles (e.g., *independent components*); the roman terms are architectures (e.g., communicating system). There are specific instances of the architectures (e.g., a client-server architecture is a distributed system). But these notions are not firm!

Data-flow systems: Batch-sequential and Pipe-and-filter



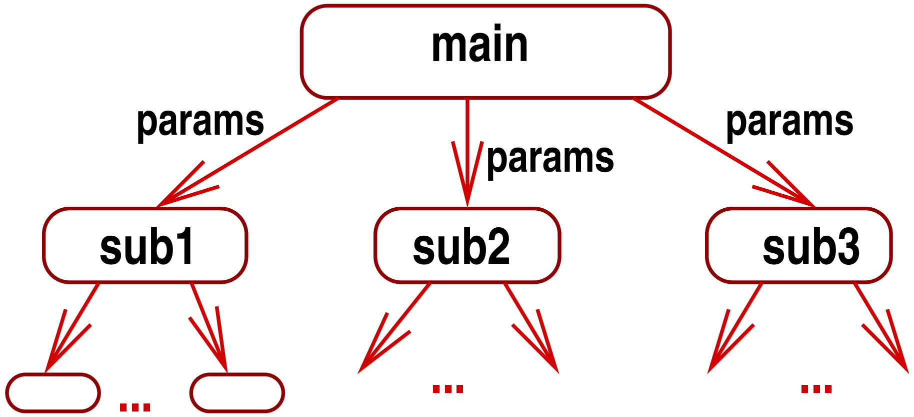
text → Scan → tokens → Parse → tree → GenCode → code

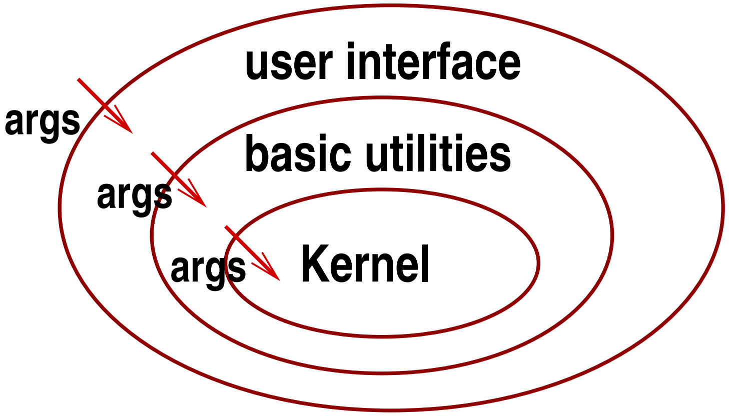
	Batch sequential	*Pipe and filter*
Components:	whole program	filter (function)
Connectors:	conventional input-output	pipe (data flow)
Constraints:	components execute to completion, consuming entire input, producing entire output	data arrives in increments to filters

Examples: Unix shells, signal processing, multi-pass compilers

Advantages: easy to unplug and replace filters; interactions between components easy to analyze. *Disadvantages:* interactivity with end-user severely limited; performs as quickly as slowest component.

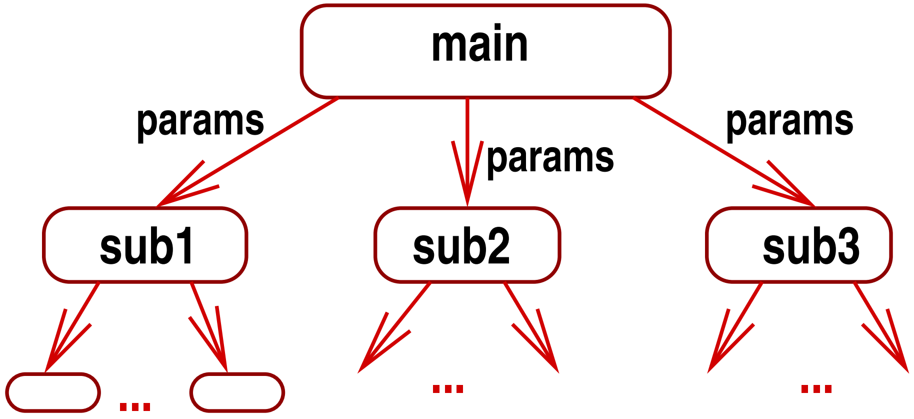
Call-and-return systems: subroutine and layered

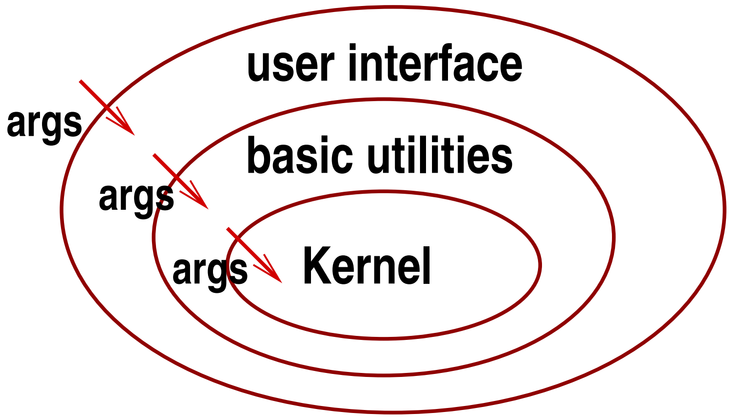




	Subroutine	*Layered*
Components:	subroutines (“servers”)	functions (“servers”)
Connectors:	parameter passing	protocols
Constraints:	hierarchical execution and encapsulation	functions within a layer invoke (API of) others at next lower layer

Examples: modular, object-oriented, N-tier systems (subroutine); communication protocols, operating systems (layered)

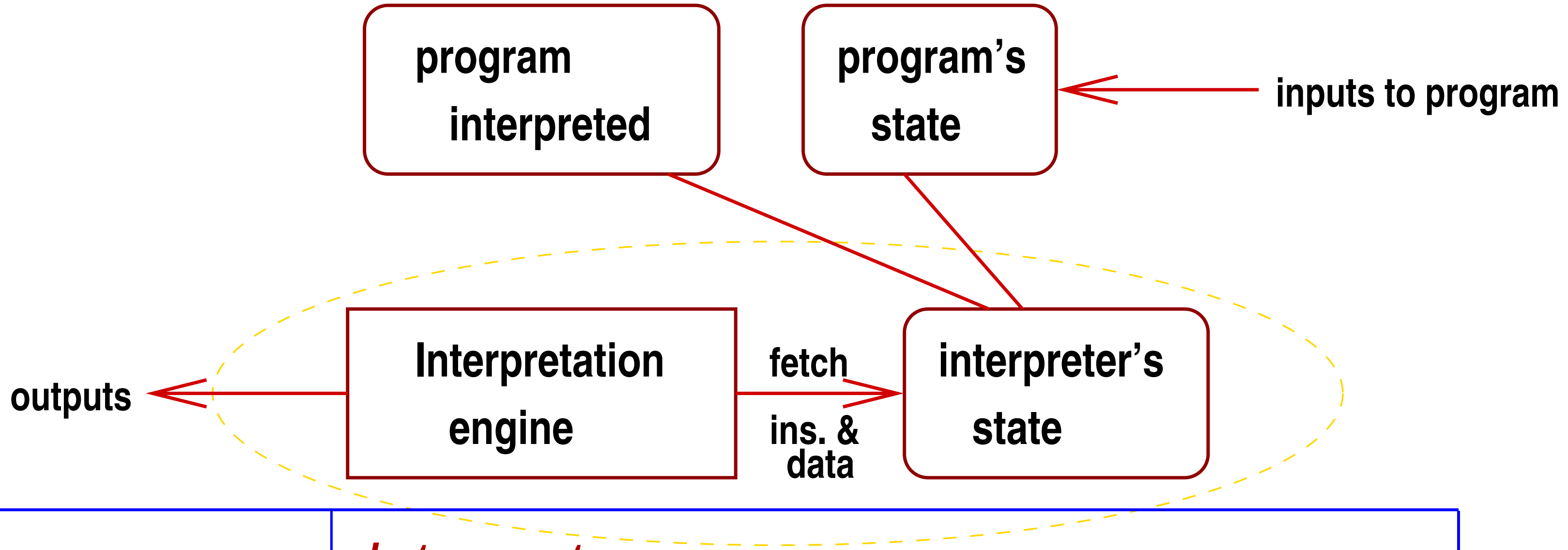




Advantages: hierarchical decomposition of solution; limits range of interactions between components, simplifying correctness reasoning; each layer defines a *virtual machine*; supports portability (by replacing lowest-level components).

Disadvantages: components must know the identities of other components to connect to them; side effects complicate correctness reasoning (e.g., `A` uses `C`, `B` uses and changes `C`, the result is an unexpected side effect from `A`'s perspective; components sensitive to performance at lower levels/layers.

Virtual machine: interpreter

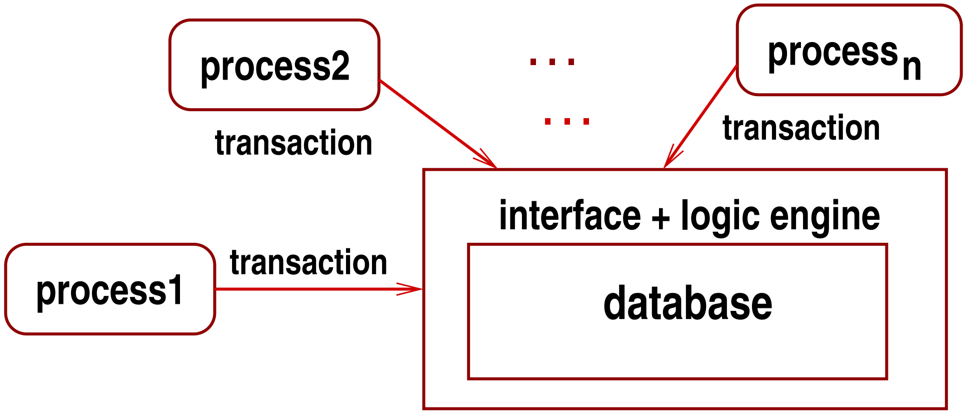


	Interpreter
Components:	"memories" and state-machine engine
Connectors:	fetch and store operations
Constraints:	engine's "execution cycle" controls the simulation of program's execution

Examples: high-level programming-language interpreters, byte-code machines, virtual machines

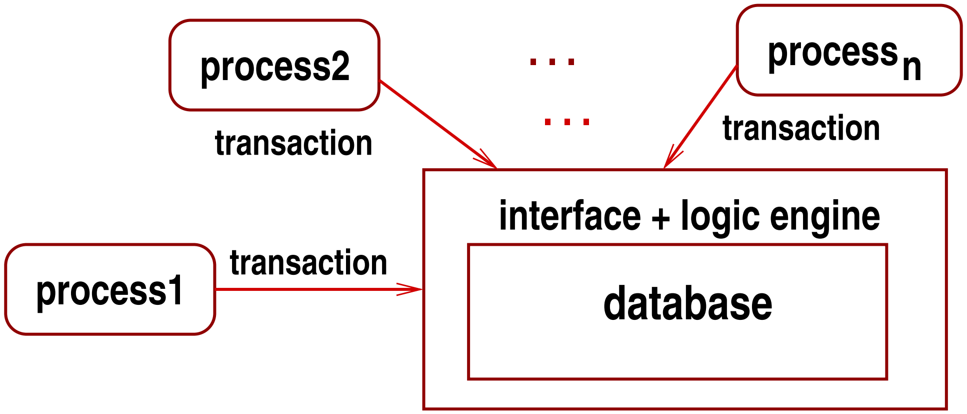
Advantages: rapid prototyping *Disadvantages:* inefficient.

Repositories: databases and blackboards



	Database	*Blackboard*
Components:	processes and database	knowledge sources and blackboard
Connectors:	queries and updates	notifications and updates
Constraints:	transactions (queries and updates) drive computation	knowledge sources respond when enabled by the state of the blackboard. Problem is solved by cooperative computation on blackboard.

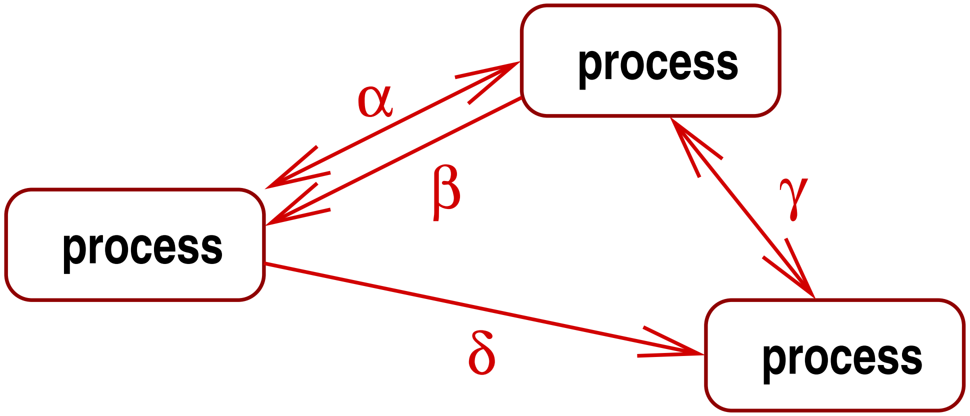
Examples: speech and pattern recognition (blackboard); syntax editors and compilers (parse tree and symbol table are repositories)



Advantages: easy to add new processes.

Disadvantages: alterations to repository affect all components.

Independent components: communicating processes

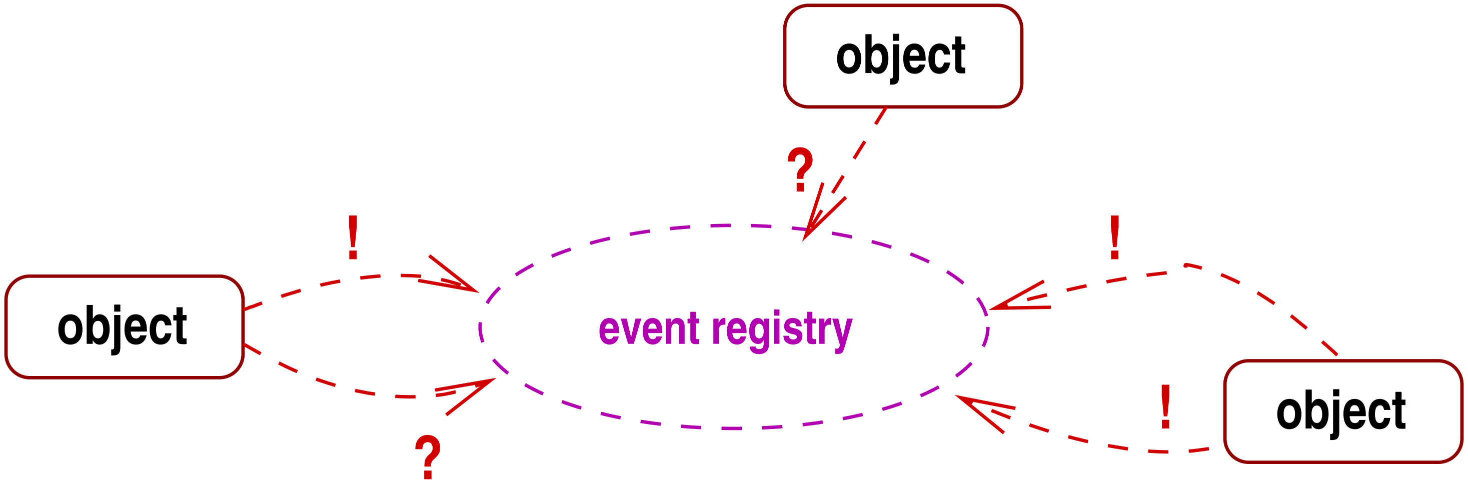


	Communicating processes
Components:	processes (“tasks”)
Connectors:	ports or buffers or RPC
Constraints:	processes execute in parallel and send messages (synchronously or asynchronously)

Example: client-server and peer-to-peer architectures

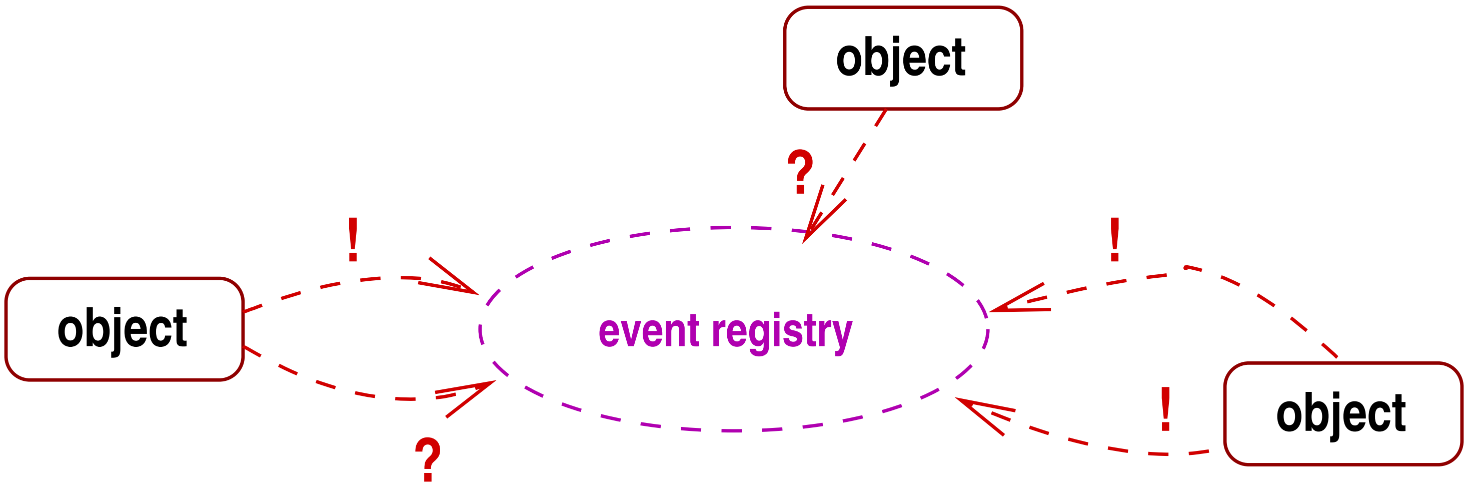
Advantages: easy to add and remove processes. *Disadvantages:* difficult to reason about control flow.

Independent components: event systems



	Event systems
Components:	objects or processes (“threads”)
Connectors:	event broadcast and notification (implicit invocation)
Constraints:	components “register” to receive event notification; components signal events, environment notifies registered “listeners”

Examples: GUI-based systems, debuggers, syntax-directed editors, database consistency checkers



Advantages: easy for new listeners to register and unregister dynamically; component reuse.

Disadvantages: difficult to reason about control flow and to formulate system-wide invariants of correct behavior.

Three architectures for a compiler (Garlan and Shaw)

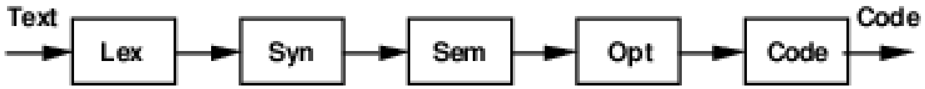


Figure 15: Traditional Compiler Model

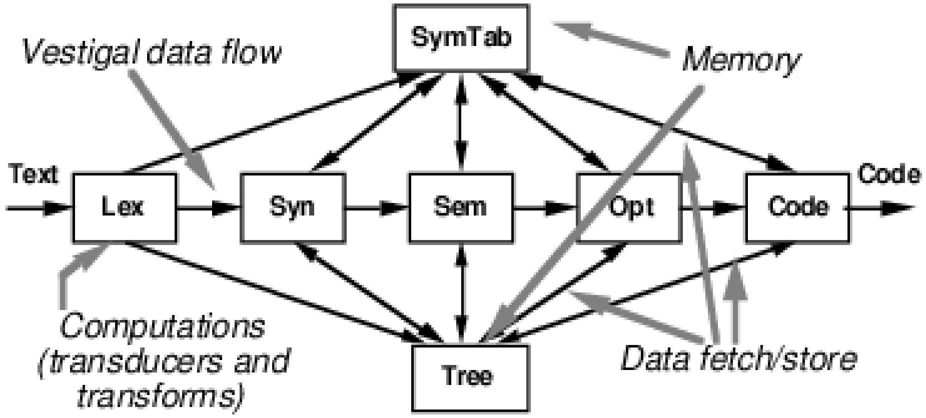


Figure 17: Modern Canonical Compiler

The symbol table and tree are “shared-data connectors”

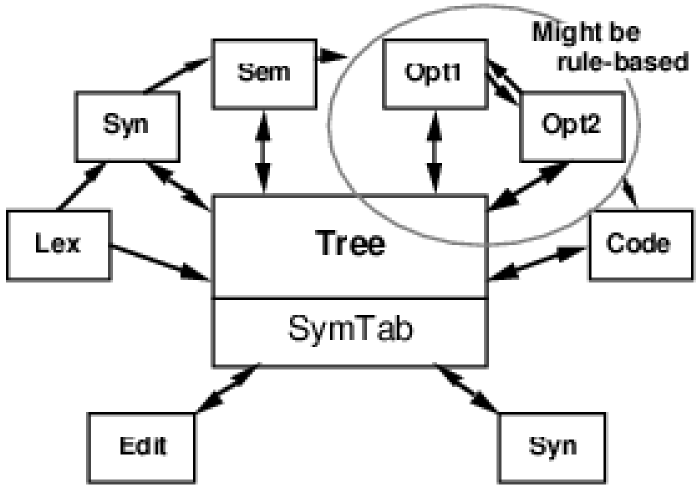


Figure 18: Canonical Compiler, Revisited

The blackboard triggers incremental checking and code generation

What do we gain from using a software architecture?

1. the architecture helps us *communicate* the system’s design to the project’s stakeholders (users, managers, implementors)
2. it helps us *analyze* design decisions
3. it helps us *reuse* concepts and components in future systems

4. Architecture Description Languages

A language for connectors: UniCon

Shaw developed a language, *UniCon* (*Universal Connector Language*), for describing connectors and components.

Components **are specified by** ***interfaces*****, which include**

(i) type;

(ii) attributes with values that specialize the type;

(iii) players, which are the component's connection points. Each player is itself typed.

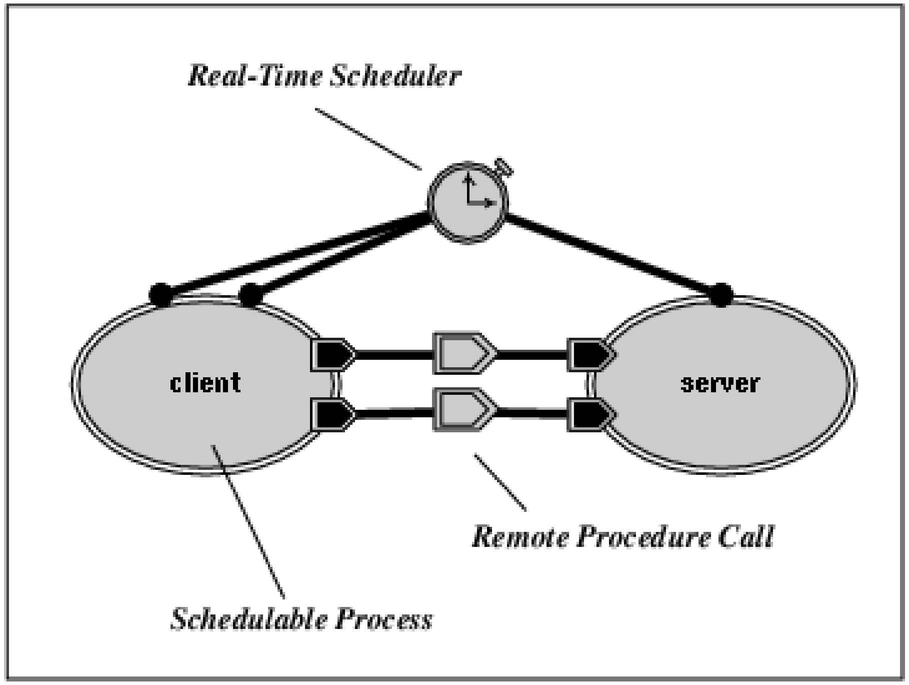
Connectors **are specified by** ***protocols*****; they have**

(i) type;

(ii) specific properties that specialize the type;

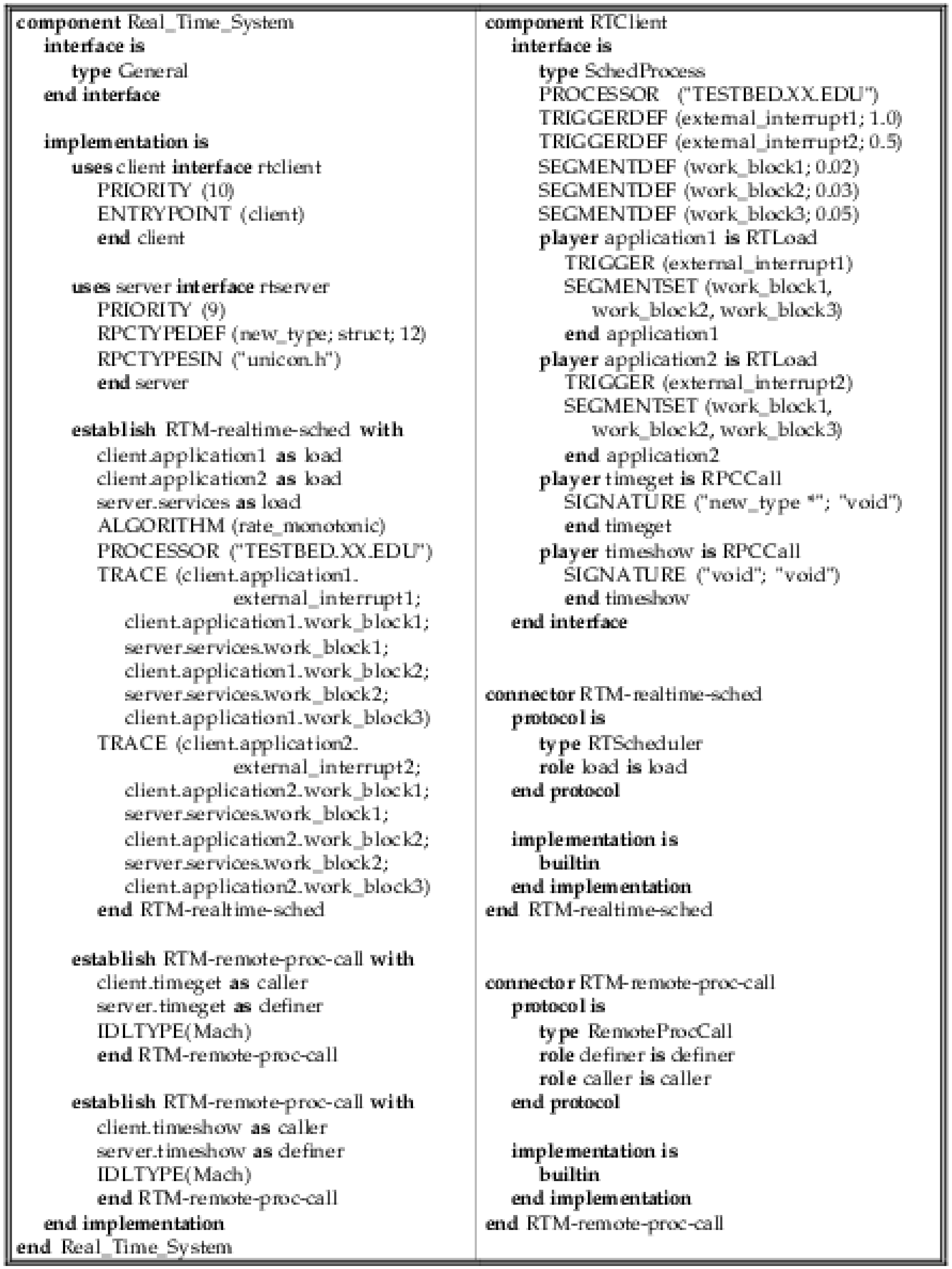
(iii) roles that the connector uses to mediate (make) communication between components.

Graphical depiction of an assembly of three components and four connectors:



A development tool helps the designer draw the configuration and map it to coding.

Reference: M. Shaw, R. DeLine, and G. Zelesnik, Abstractions and Implementations for Architectural Connections. In 3d Int. Conf. Configurable Distributed Systems, Annapolis, Maryland, May 1996.



```
component Real_Time_System
    interface is
        type General
    end interface

    implementation is
        uses client interface rtclient
            PRIORITY (10)
            ENTRYPOINT (client)
            end client

        uses server interface rtserver
            PRIORITY (9)
            RPCTYPEDEF (new_type; struct; 12)
            RPCTYPESIN ("unicon.h")
            end server

        establish RTM-realtime-sched with
            client.application1 as load
            client.application2 as load
            server.services as load
            ALGORITHM (rate_monotonic)
            PROCESSOR ("TESTBED.XX.EDU")
            TRACE (client.application1.
                            external_interrupt1;
                client.application1.work_block1;
                server.services.work_block1;
                client.application1.work_block2;
                server.services.work_block2;
                client.application1.work_block3)
            TRACE (client.application2.
                            external_interrupt2;
                client.application2.work_block1;
                server.services.work_block1;
                client.application2.work_block2;
                server.services.work_block2;
                client.application2.work_block3)
            end RTM-realtime-sched

        establish RTM-remote-proc-call with
            client.timeget as caller
            server.timeget as definer
            IDLTYPE(Mach)
            end RTM-remote-proc-call

        establish RTM-remote-proc-call with
            client.timeshow as caller
            server.timeshow as definer
            IDLTYPE(Mach)
            end RTM-remote-proc-call
    end implementation
end Real_Time_System
```

```
component RTClient
    interface is
        type SchedProcess
        PROCESSOR   ("TESTBED.XX.EDU")
        TRIGGERDEF (external_interrupt1; 1.0)
        TRIGGERDEF (external_interrupt2; 0.5)
        SEGMENTDEF (work_block1; 0.02)
        SEGMENTDEF (work_block2; 0.03)
        SEGMENTDEF (work_block3; 0.05)
        player application1 is RTLoad
            TRIGGER (external_interrupt1)
            SEGMENTSET (work_block1,
                work_block2, work_block3)
            end application1
        player application2 is RTLoad
            TRIGGER (external_interrupt2)
            SEGMENTSET (work_block1,
                work_block2, work_block3)
            end application2
        player timeget is RPCCall
            SIGNATURE ("new_type *"; "void")
            end timeget
        player timeshow is RPCCall
            SIGNATURE ("void"; "void")
            end timeshow
    end interface


connector RTM-realtime-sched
    protocol is
        type RTScheduler
        role load is load
    end protocol

    implementation is
        builtin
    end implementation
end RTM-realtime-sched


connector RTM-remote-proc-call
    protocol is
        type RemoteProcCall
        role definer is definer
        role caller is caller
    end protocol

    implementation is
        builtin
    end implementation
end RTM-remote-proc-call
```

uses statements instantiate the parts composed

connect statements state how players satisfy roles

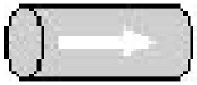
bind statements map the external interface to the internal configuration

Connectors described in UniCon:

- data-flow connectors (pipe)
- procedural connectors (procedure call, remote procedure call): pass control
- data-sharing connectors (data access): export and import data
- resource-contention connectors (RT scheduler): competition for resources
- aggregate connectors (PL bundler): compound connections

Pipe Connector

Informal Description: The Unix abstraction for pipe, i.e. a bounded queue of bytes that are produced at a source and consumed at a sink. Also supports interactions between pipes and files, choosing the correct Unix implementation.



Icon: pipe section

Properties: *PipeType*, the kind of Unix pipe. Possible values Named, Unnamed

Roles: *Source*

Description: the source end of the pipe
Accepts player types: *StreamOut* of component Filter; *ReadNext* of component SeqFile
Properties: *MinConns*, minimum number of connections. Integer values, default 1
MaxConns, maximum number of connections. Integer values, default 1

Sink

Description: the sink end of the pipe
Accepts player types: *StreamIn* of component Filter; *WriteNext* of component SeqFile
Properties: *MinConns, MaxConns*, as for Source

ProcedureCall Connector

Informal Description: The architectural abstraction corresponding to the procedure call of standard programming languages. Requires signatures (eventually pre/post conditions) in the RoutineDef and RoutineCall players to match; if they don't, requests remediation. Supports renaming.

Icon: blunt arrowhead

Roles: *Definer*

Description: role played by the procedure definition
Accepts player types: *RoutineDef* of component Computation or Module
Properties: *MinConns*, minimum number of definitions allowed. Integer, must be 1
MaxConns, maximum number of definitions allowed. Integer, must be 1

Caller

Description: the role played by the procedure call
Accepts player types: *RoutineCall* of component Computation or Module
Properties: *MinConns*, minimum number of callers allowed. Integer, default 1
MaxConns, maximum number of callers allowed. Integer, default many

RemoteProcCall Connector

Informal Description: The abstraction for the remote procedure call facility supplied by the operating system. Requires signatures and eventually pre/post conditions in the RPCDef and RPCCall players to match. RemoteProcCall connectors require much more UniCon support than ProcedureCall connectors, as they must establish communication paths between processes.

Icon: bordered blunt arrowhead

Roles: *Definer*

Description: role played by the procedure definition
Accepts player types: *RPCDef* of component Process or SchedProcess
Properties: *MinConns, MaxConns*, as for ProcedureCall

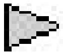
Caller

Description: the role played by the procedure call
Accepts player types: *RPCCall* of component Process or SchedProcess
Properties: *MinConns, MaxConns*, as for ProcedureCall

DataAccess Connector

Informal Description: The architectural abstraction corresponding to imported and exported data of conventional programming languages.

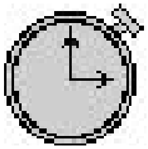
Icon: triangle



Roles: *Definer*, essentially similar to *Definer* of *ProcedureCall*
User, essentially similar to *Caller* of *ProcedureCall*

RTScheduler Connector

Informal Description: Mediates competition for processor resources among a set of real-time processes (requires an operating system with appropriate real-time capabilities).



Icon: stopwatch

Properties: *Algorithm*, the scheduling discipline. Possible values: RateMonotonic, TimeSharing, EarliestDeadline, DeadlineMonotonic, RoundRobinFixPriority, FIFOFixPriority
Processor, the name of the processor on which this set of processes will run
Trace, a path through the real-time code and the trigger that invokes it

Roles: *Load*

Description: the role played by a real-time load on a processor
Accepts player types: *RTLoad* of component SchedProcess
Properties: *MinConns*, minimum number of competing processes. Integer, default 2
MaxConns, maximum number of competing processes. Integer, default many

PLBundler Connector

Informal Description: A composite abstraction for matching definitions and uses of a collection of procedures and data. It allows multiple procedure and data definitions and uses to be matched with a single abstraction. Supports renaming.

Icon: chain links

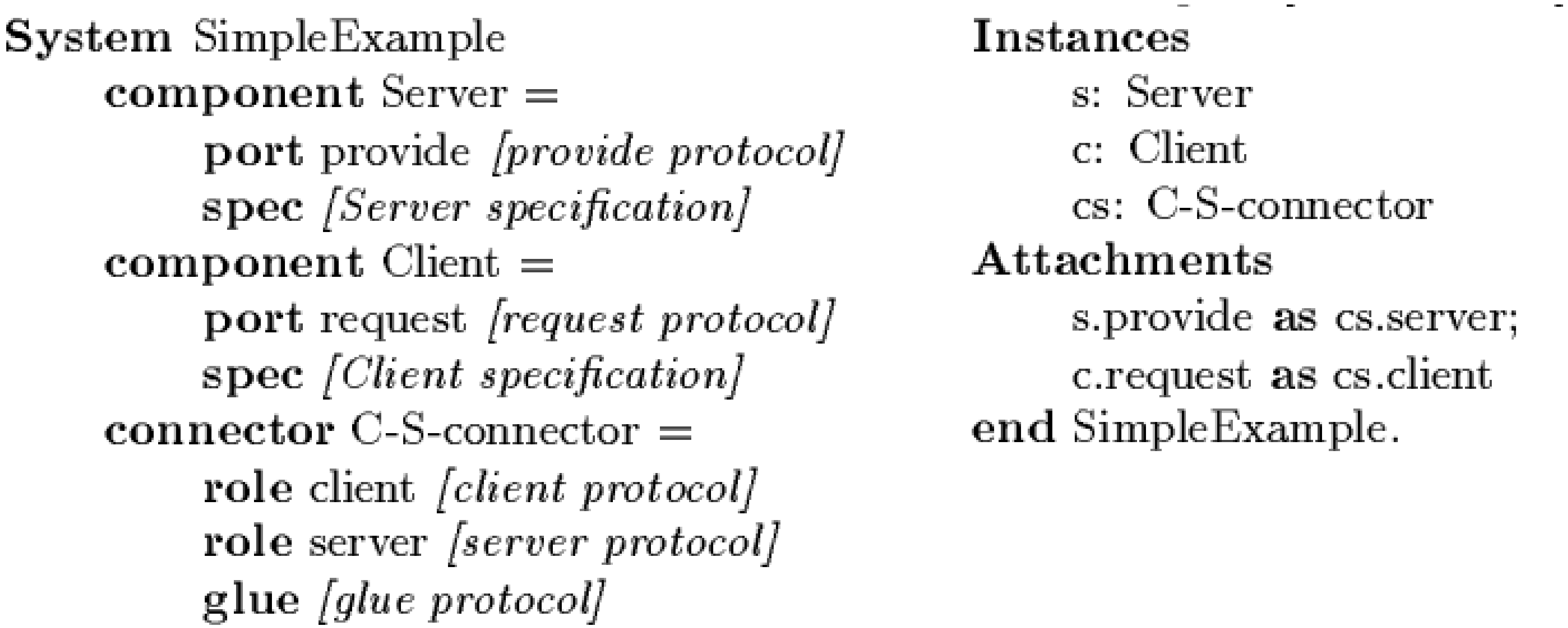
Properties: *Match*, the correspondences between individual definitions in the bundles. Values are sets of pairs of names.

Roles: *Participant*

> **Description:** a set of definitions and uses to take part in the linkage
> **Accepts player types:** *PLBundle* of component Computation, Module, or SharedData
> **Properties:** *MinConns*, minimum number of bundles to match. Integer, default 2
> *MaxConns*, maximum number of bundles to match. Integer, default many

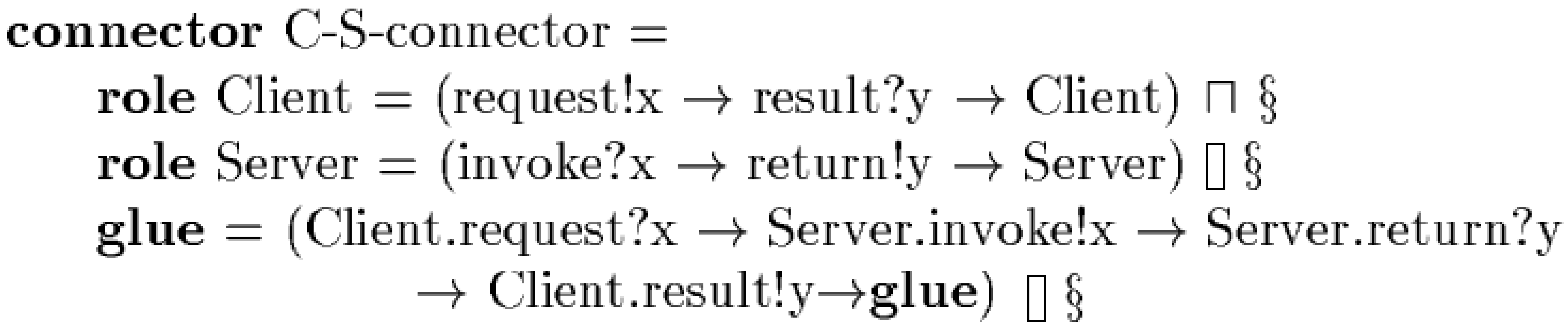
Wright: Unicon + CSP

Garlan and Allen developed Wright to specify protocols. Here is a single-client/single-server example:



```
System SimpleExample
    component Server =
        port provide [provide protocol]
        spec [Server specification]
    component Client =
        port request [request protocol]
        spec [Client specification]
    connector C-S-connector =
        role client [client protocol]
        role server [server protocol]
        glue [glue protocol]
Instances
    s: Server
    c: Client
    cs: C-S-connector
Attachments
    s.provide as cs.server;
    c.request as cs.client
end SimpleExample.
```

The *protocols* are specified with Hoare’s CSP (Communicating Sequential Processes) algebra.



connector C-S-connector =

role Client = (request!x → result?y → Client) ⊓ §

role Server = (invoke?x → return!y → Server) □ §

glue = (Client.request?x → Server.invoke!x → Server.return?y → Client.result!y→**glue**) □ §

The *glue* protocol synchronizes the Client and Server roles:

$$\texttt{Client} \parallel \texttt{Server} \parallel \texttt{glue}$$

$$\Rightarrow \texttt{result?y} \longrightarrow \texttt{Client} \parallel \texttt{Server} \parallel \texttt{Server.invoke!x} \longrightarrow \texttt{...}$$

$$\Rightarrow \texttt{result?y} \longrightarrow \texttt{Client} \parallel \texttt{return!y} \longrightarrow \texttt{Server} \parallel$$

$$\texttt{Server.return?y} \longrightarrow \texttt{...}$$

$$\Rightarrow \texttt{...} \quad \Rightarrow \texttt{Client} \parallel \texttt{Server} \parallel \texttt{glue}$$

Forms of CSP processes:

- prefixing: $e \longrightarrow P$

  $plusOne?x \to return!x + 1 \to \cdots \parallel plusOne!2 \to return?y \to \cdots$

  $\Rightarrow \; return!2 + 1 \to \cdots \parallel return?y \to \cdots$

- external choice: $P \Box Q$

  $plusOne?x \to \cdots \; \Box \; plusTwo?x \to \cdots x + 2 \cdots \parallel plusTwo!5 \to \cdots$

  $\Rightarrow \; \cdots 5 + 2 \cdots \parallel \; \cdots$

- internal choice: $P \sqcap Q$

  $plusOne?x \to \cdots \parallel plusOne!5 \to \cdots \sqcap plusTwo!5 \to \cdots$

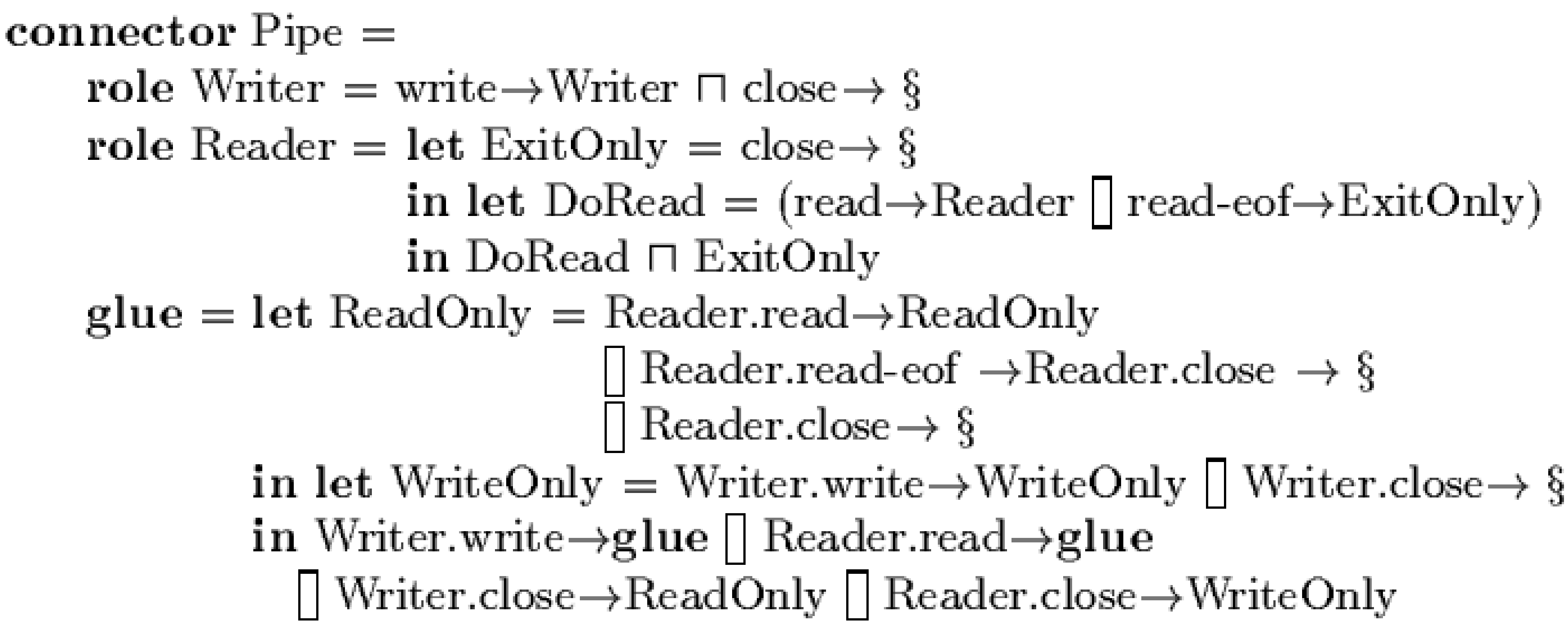
  $\Rightarrow \; plusOne?x \to \cdots \parallel plusTwo!5 \to \cdots$

- parallel composition: $P \| Q$

- halt: §

- (tail) recursion: $p = \cdots p$ (More formally, $\mu z.P$, where z may occur free in P.)

A pipe protocol in Wright

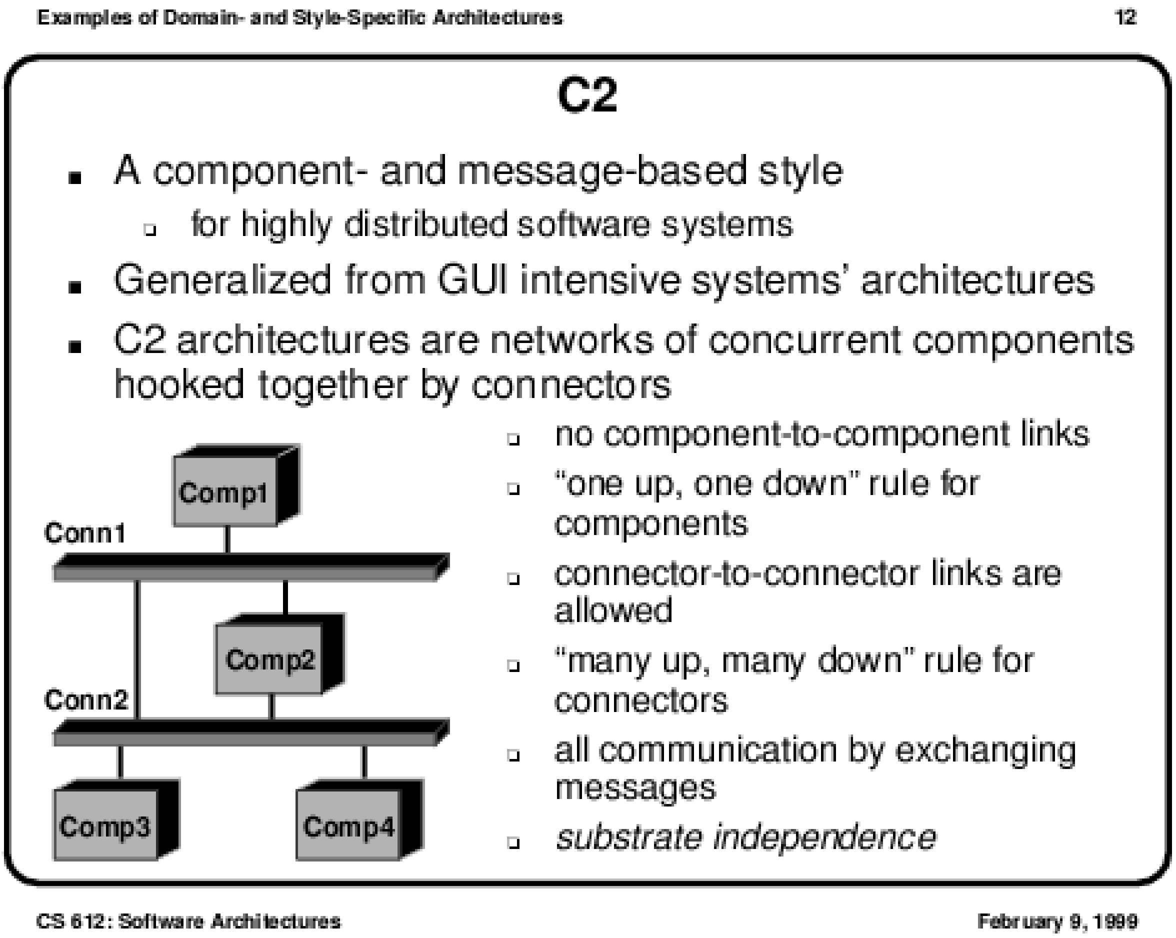


```
connector Pipe =
    role Writer = write→Writer ⊓ close→ §
    role Reader = let ExitOnly = close→ §
                  in let DoRead = (read→Reader □ read-eof→ExitOnly)
                  in DoRead ⊓ ExitOnly
    glue = let ReadOnly = Reader.read→ReadOnly
                        □ Reader.read-eof →Reader.close → §
                        □ Reader.close→ §
         in let WriteOnly = Writer.write→WriteOnly □ Writer.close→ §
         in Writer.write→glue □ Reader.read→glue
           □ Writer.close→ReadOnly □ Reader.close→WriteOnly
```

Reference: R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM 1997.*

C2: an N-tier framework and language

Developed at Univ. of California, Irvine, Institute of Software Research: `http://www.isr.uci.edu/architecture/c2.html`



Examples of Domain- and Style-Specific Architectures 12

C2

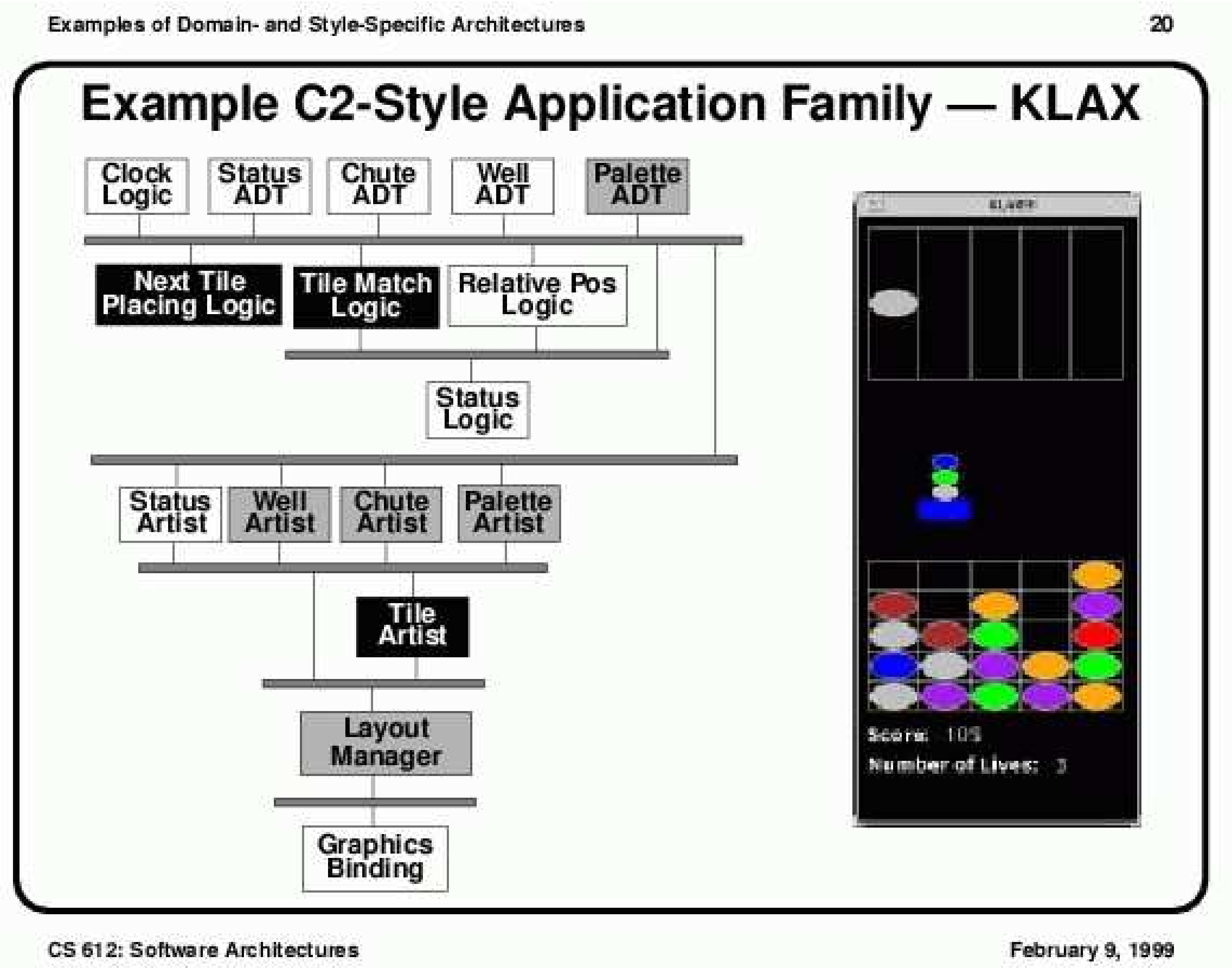
- A component- and message-based style
  - for highly distributed software systems
- Generalized from GUI intensive systems' architectures
- C2 architectures are networks of concurrent components hooked together by connectors
  - no component-to-component links
  - "one up, one down" rule for components
  - connector-to-connector links are allowed
  - "many up, many down" rule for connectors
  - all communication by exchanging messages
  - *substrate independence*

CS 612: Software Architectures February 9, 1999

Diagrams are from Medvidovic's course,

`http://sunset.usc.edu/classes/cs578_2002`

Example architecture in C2: video game



Examples of Domain- and Style-Specific Architectures 20

Example C2-Style Application Family — KLAX

Clock Logic

Status ADT

Chute ADT

Well ADT

Palette ADT

Next Tile Placing Logic

Tile Match Logic

Relative Pos Logic

Status Logic

Status Artist

Well Artist

Chute Artist

Palette Artist

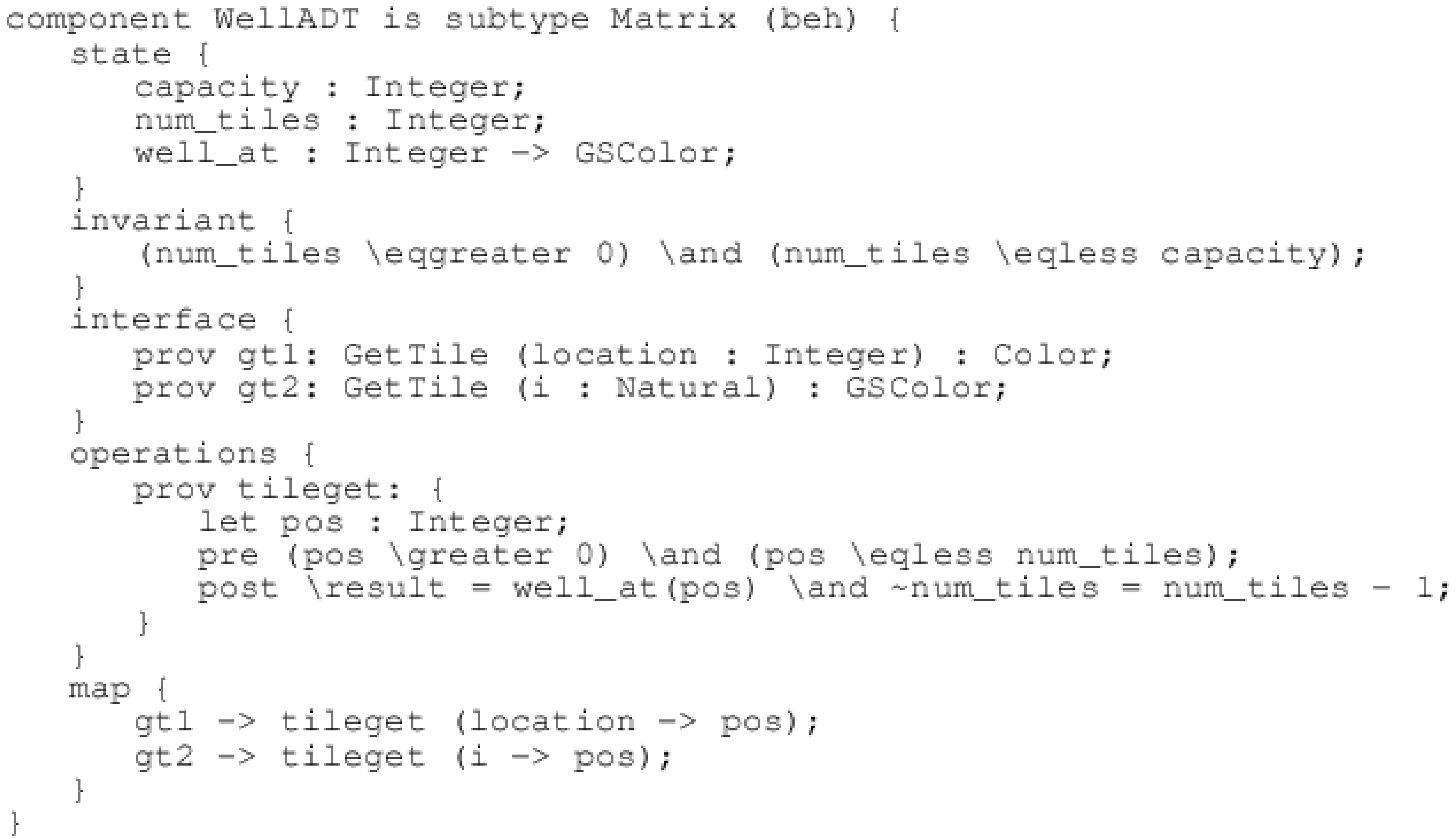
Tile Artist

Layout Manager

Graphics Binding

CS 612: Software Architectures February 9, 1999

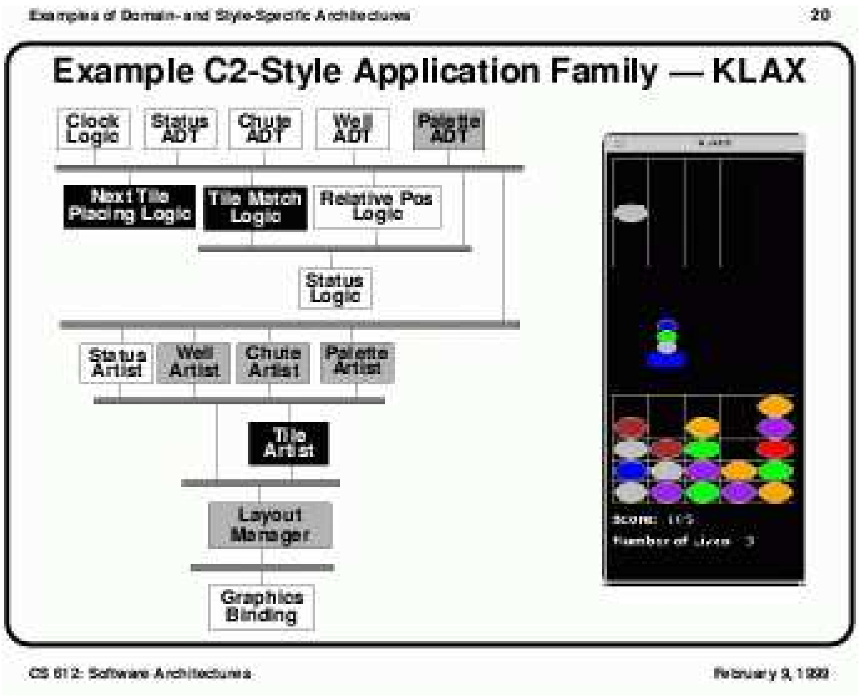
Here is a *C2SADEL* description of the video game’s “Well” component:



```
component WellADT is subtype Matrix (beh) {
   state {
      capacity : Integer;
      num_tiles : Integer;
      well_at : Integer -> GSColor;
   }
   invariant {
      (num_tiles \eqgreater 0) \and (num_tiles \eqless capacity);
   }
   interface {
      prov gt1: GetTile (location : Integer) : Color;
      prov gt2: GetTile (i : Natural) : GSColor;
   }
   operations {
      prov tileget: {
         let pos : Integer;
         pre (pos \greater 0) \and (pos \eqless num_tiles);
         post \result = well_at(pos) \and ~num_tiles = num_tiles - 1;
      }
   }
   map {
      gt1 -> tileget (location -> pos);
      gt2 -> tileget (i -> pos);
   }
}
```

Reference: N. Medvidovic, et al. A Language and Environment for Architecture-Based Software Development and Evolution. 21st Int. Conf. on Software Engineering, Los Angeles, May 1999.

And here is a description of a connector and part of the configuration:



Examples of Domain- and Style-Specific Architectures 20

Example C2-Style Application Family — KLAX

Clock Logic

Status ADT

Chute ADT

Well ADT

Palette ADT

Next Tile Placing Logic

Tile Match Logic

Relative Pos Logic

Status Logic

Status Artist

Well Artist

Chute Artist

Palette Artist

Tile Artist

Layout Manager

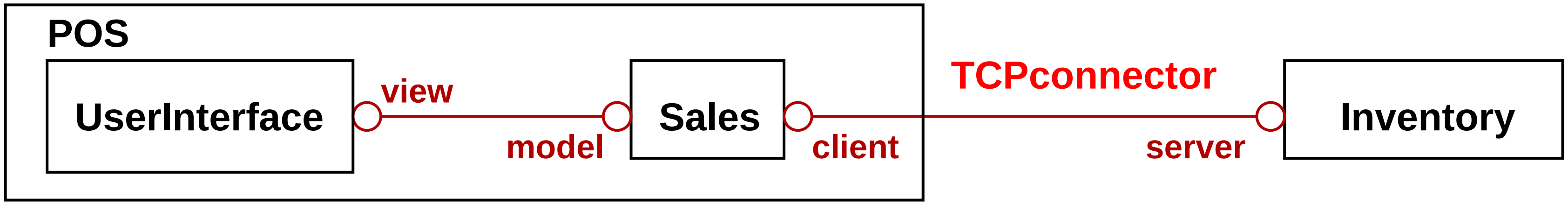
Graphics Binding

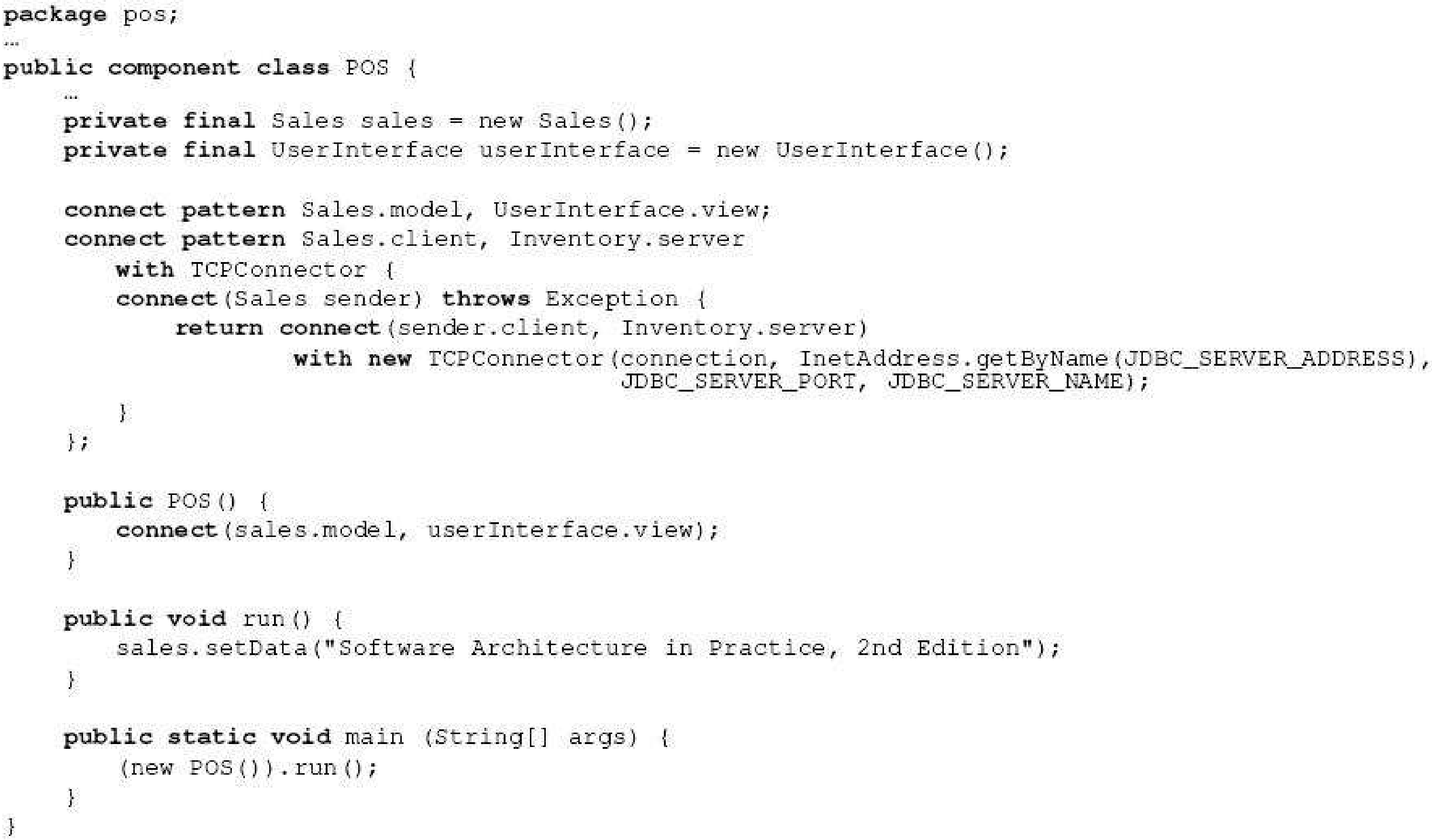
CS 612: Software Architectures February 9, 1999

```
connector BroadcastConn is {
   message_filter no_filtering;
}


architectural_topology {
   component_instances {
      Well : WellADT;
      WellArt : WellArtist;
      MatchLogic : TileMatchLogic;
   }
   connector_instances {
      ADTConn : BroadcastConn;
      ArtConn : BroadcastConn;
   }
   connections {
      connector ADTConn {
         top Well;
         bottom MatchLogic, ArtConn;
      }
      connector ArtConn {
         top ADTConn;
         bottom WellArt;
      }
   }
}
```

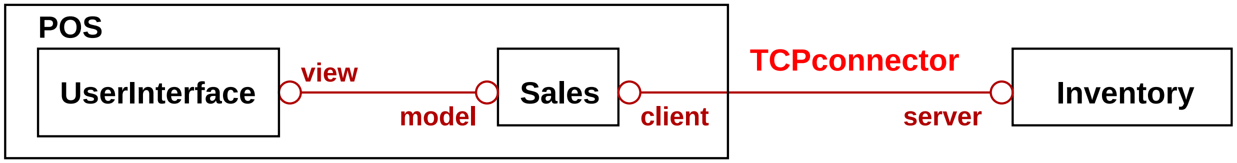
ArchJava: Java extended with Unicon features

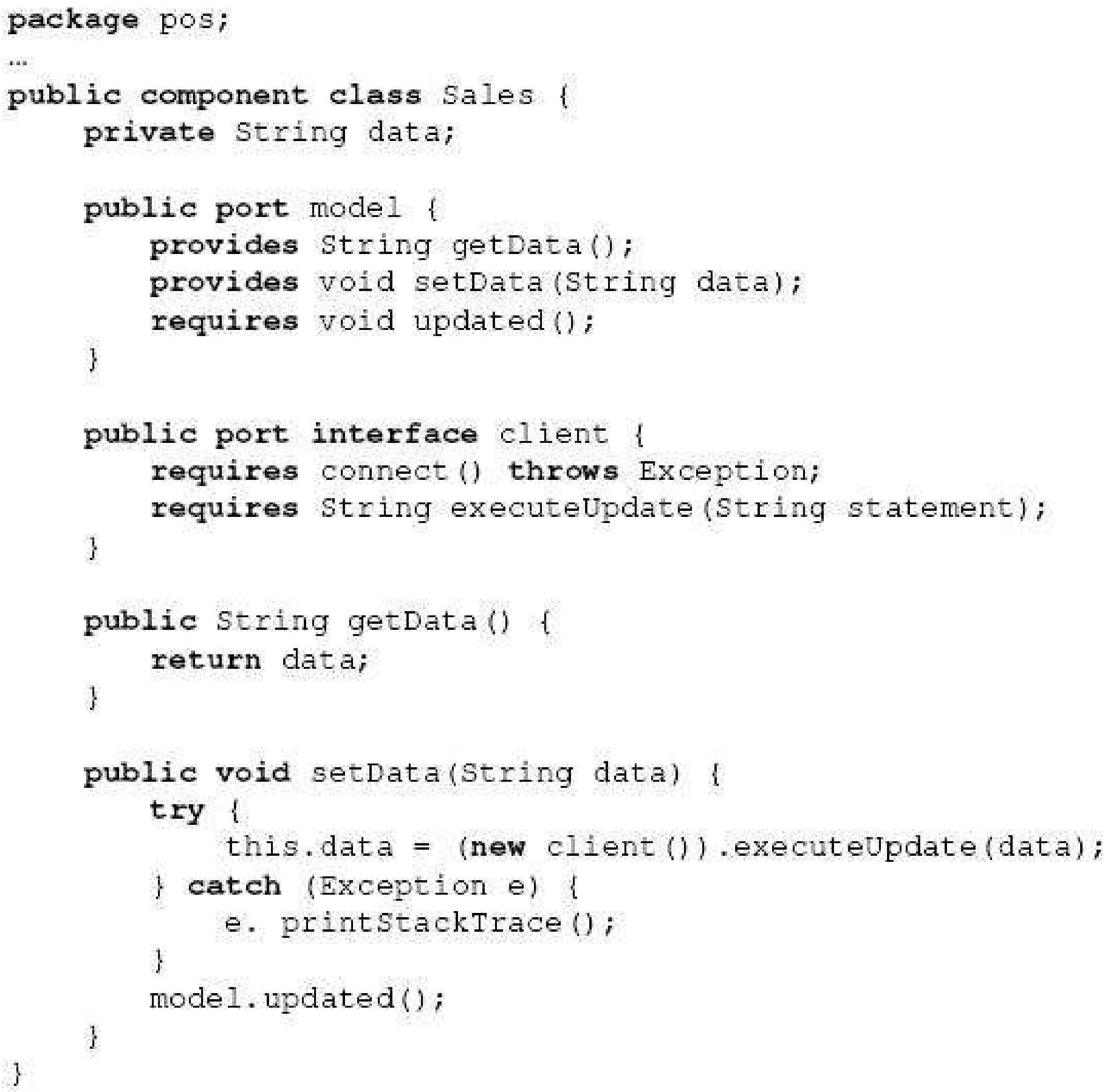
- Each component (class) has its own interfaces (*ports*) that list which methods it `requires` and `provides`
- Connectors are coded as classes, too, and extend the basic classes, `Connector`, `Port`, `Method`, etc.
- The ArchJava run-time platform includes a run-time type checker that enforces correctness of run-time connections (e.g., RPC, TCP)
- There is an open-source implementation and Eclipse plug-in
- `www.archjava.org`



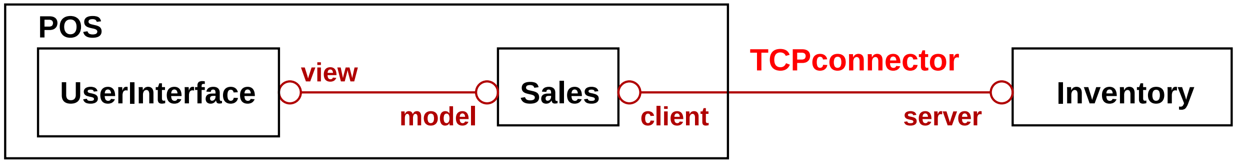


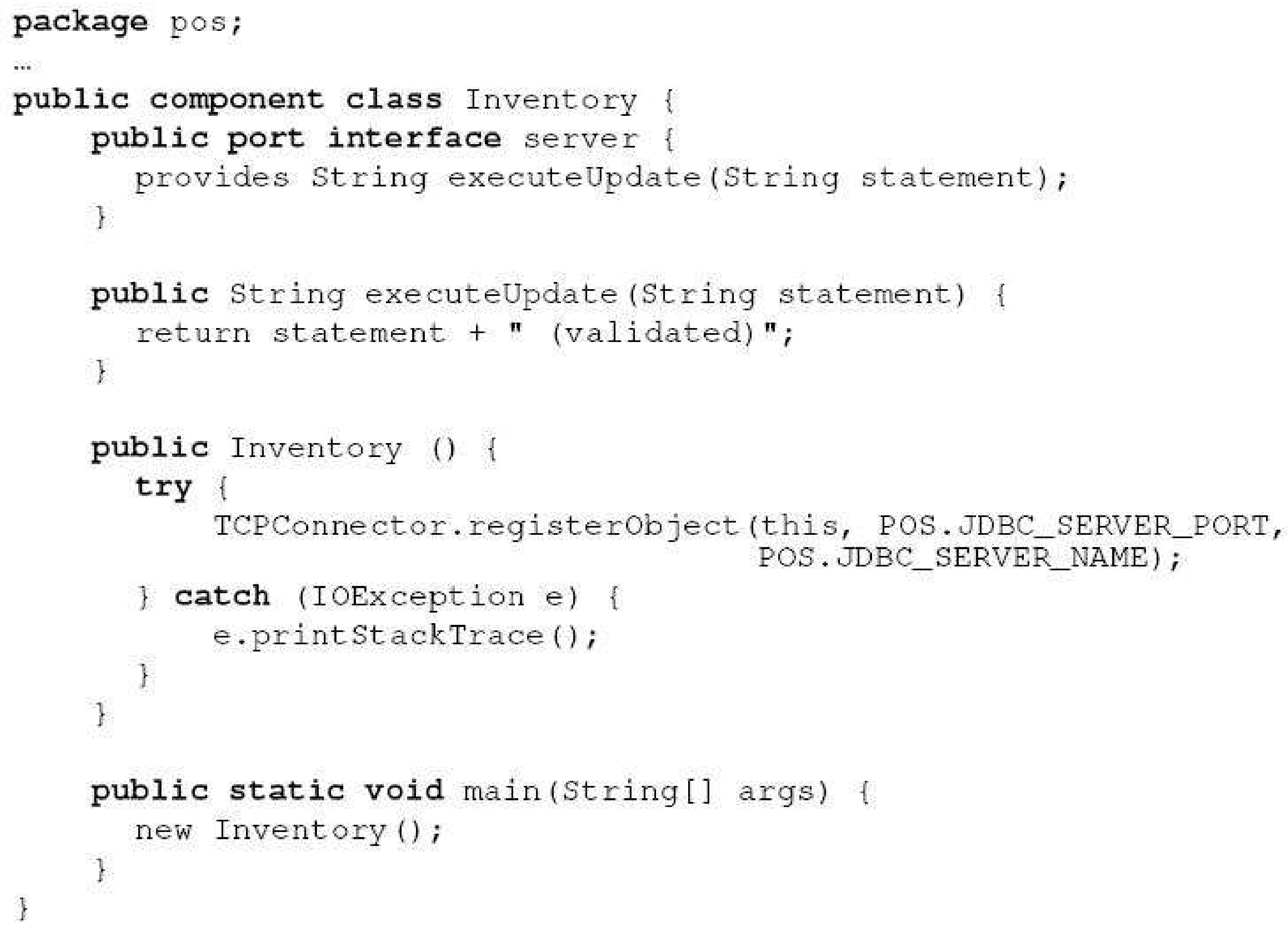
```
package pos;
…
public component class POS {
    …
    private final Sales sales = new Sales();
    private final UserInterface userInterface = new UserInterface();

    connect pattern Sales.model, UserInterface.view;
    connect pattern Sales.client, Inventory.server
        with TCPConnector {
        connect(Sales sender) throws Exception {
            return connect(sender.client, Inventory.server)
                    with new TCPConnector(connection, InetAddress.getByName(JDBC_SERVER_ADDRESS),
                                                JDBC_SERVER_PORT, JDBC_SERVER_NAME);
        }
    };

    public POS() {
        connect(sales.model, userInterface.view);
    }

    public void run() {
        sales.setData("Software Architecture in Practice, 2nd Edition");
    }

    public static void main (String[] args) {
        (new POS()).run();
    }
}
```





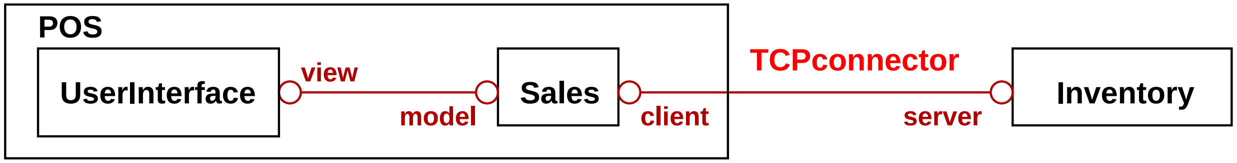
```
package pos;
...
public component class Sales {
    private String data;

    public port model {
        provides String getData();
        provides void setData(String data);
        requires void updated();
    }

    public port interface client {
        requires connect() throws Exception;
        requires String executeUpdate(String statement);
    }

    public String getData() {
        return data;
    }

    public void setData(String data) {
        try {
            this.data = (new client()).executeUpdate(data);
        } catch (Exception e) {
            e. printStackTrace();
        }
        model.updated();
    }
}
```

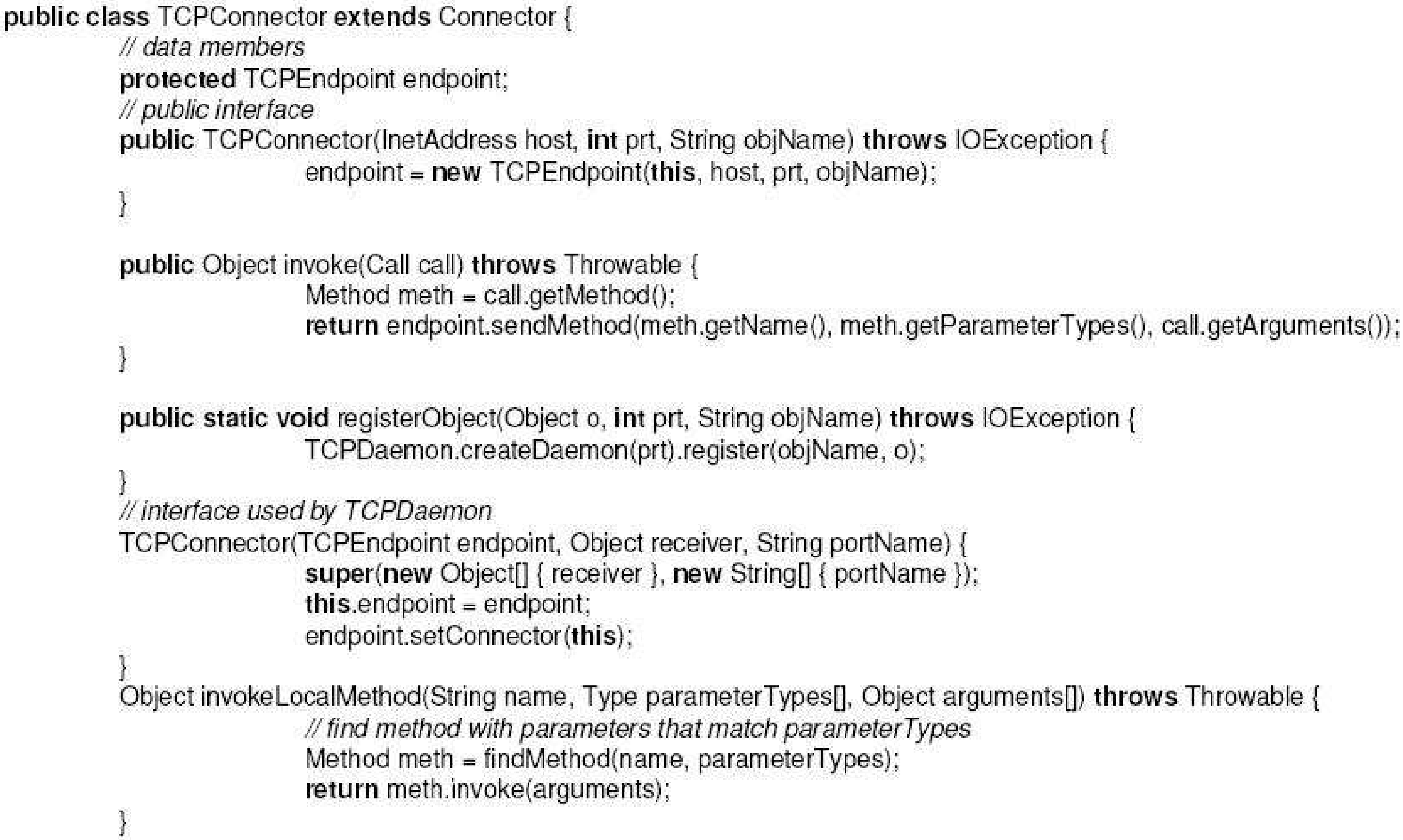




```
package pos;
...
public component class Inventory {
    public port interface server {
      provides String executeUpdate(String statement);
    }

    public String executeUpdate(String statement) {
      return statement + " (validated)";
    }

    public Inventory () {
      try {
          TCPConnector.registerObject(this, POS.JDBC_SERVER_PORT,
                                          POS.JDBC_SERVER_NAME);
      } catch (IOException e) {
          e.printStackTrace();
      }
    }

    public static void main(String[] args) {
      new Inventory();
    }
}
```

From K. M. Hansen, `www.daimi.dk/~marius/teaching/ATiSA2005`





```
public class TCPConnector extends Connector {
        // data members
        protected TCPEndpoint endpoint;
        // public interface
        public TCPConnector(InetAddress host, int prt, String objName) throws IOException {
                        endpoint = new TCPEndpoint(this, host, prt, objName);
        }

        public Object invoke(Call call) throws Throwable {
                        Method meth = call.getMethod();
                        return endpoint.sendMethod(meth.getName(), meth.getParameterTypes(), call.getArguments());
        }

        public static void registerObject(Object o, int prt, String objName) throws IOException {
                        TCPDaemon.createDaemon(prt).register(objName, o);
        }
        // interface used by TCPDaemon
        TCPConnector(TCPEndpoint endpoint, Object receiver, String portName) {
                        super(new Object[] { receiver }, new String[] { portName });
                        this.endpoint = endpoint;
                        endpoint.setConnector(this);
        }
        Object invokeLocalMethod(String name, Type parameterTypes[], Object arguments[]) throws Throwable {
                        // find method with parameters that match parameterTypes
                        Method meth = findMethod(name, parameterTypes);
                        return meth.invoke(arguments);
        }
```

So, what is an architectural description language?

It is a notation (linear or graphical) for specifying an architecture.

It should specify

- *structure*: components (interfaces), connectors (protocols), configuration (both static and dynamic structure)
- *behavior*: semantical properties of individual components and connectors, patterns of acceptable communication, global invariants,
- *design patterns*: global constraints that support correctness-reasoning techniques, design- and run-time tool support, and implementation.

But it is difficult to design a *general-purpose* architectural description language that is *elegant*, *expressive*, and *useful*.

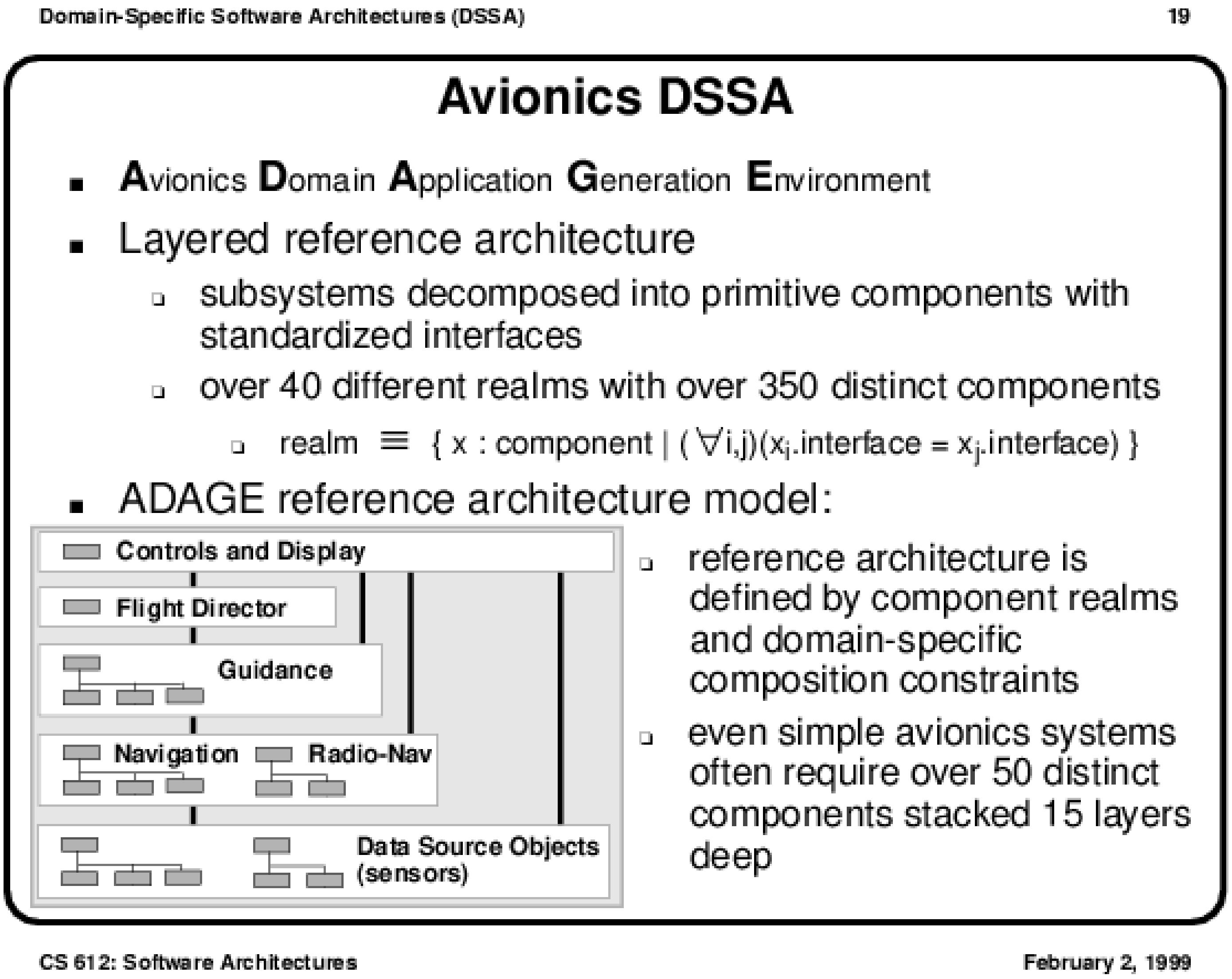
5. Domain-specific design

Domain-specific design

If the problem domain is a standard one (e.g., flight-control or telecommunications or banking), then there are precedents to follow.

A *Domain-Specific Software Architecture* has

- a *domain*: defines the problem area domain concepts and terminology; customer requirements; scenarios; configuration models (entity-relationship, data flow, etc.)
- *reference requirements*: *features* that restrict solutions to fit the domain. ("Features" are studied shortly.) Also: platform, language, user interface, security, performance
- a *reference architecture*
- a *supporting environment/infrastructure*: tools for modelling, design, implementation, evaluation; run-time platform
- a *process* or *methodology* to implement the reference architecture and evaluate it.



Domain-Specific Software Architectures (DSSA) 19

Avionics DSSA

- **A**vionics **D**omain **A**pplication **G**eneration **E**nvironment
- Layered reference architecture
  - subsystems decomposed into primitive components with standardized interfaces
  - over 40 different realms with over 350 distinct components
    - realm $\equiv$ $\{ x : \text{component} \mid (\forall i,j)(x_i.\text{interface} = x_j.\text{interface}) \}$
- ADAGE reference architecture model:

- reference architecture is defined by component realms and domain-specific composition constraints
- even simple avionics systems often require over 50 distinct components stacked 15 layers deep

CS 612: Software Architectures February 2, 1999

from Medvidovic’s course, `http://sunset.usc.edu/classes/cs578_2002`

Domain-specific (modelling) language (DSL)

is a modelling language specialized to a specific problem domain, e.g., telecommunications, banking, transportation.

We use a DSL to describe a problem and its solution in concepts familiar to people who work in the domain.

It might define (entity-relationship) models, ontologies (class hierarchies), scenarios, architectures, and implementations.

Example: a DSL for sensor-alarm networks: *domains:* sites (building, floor, hallway, room), devices (alarm, movement detector, camera, badge), people (employee, guard, police, intruder). Domain elements have *features/attributes* and *operations*. *Actions* can be by initiated by *events* — “when a movement detector detects an intruder in a room, it generates a movement-event for a camera and sends a message to a guard....”

When a DSL can generate computer implementations, it is a *domain-specific programming language*.

Domain-specific programming language

In the Unix world, these are “little languages” or “mini-languages,” designed to solve a specific class of problems. Examples are `awk`, `make`, `lex`, `yacc`, `ps`, and `Glade` (for GUI-building in X).

Other examples are Excel, HTML, XML, SQL, regular-expression notation and BNF. These are called *top-down* DSLs, because they are designed to implement domain concepts and nothing more. Non-programmers can use a top-down DSL to write solutions.

The *bottom-up* approach, called *embedded* or *in-language DSL*, starts with a dynamic-data-structure language, like Scheme or Perl or Python, and adds libraries (modules) of functions that encode domain-concepts-as-code, thus “building the language upwards towards the problem to be solved.” Experienced programmers use bottom-up DSLs to program solutions.

Tradeoffs in using (top-down) DSLs

- ✔ non-programmers can discuss and use the DSL
- ✔ the DSL supports patterns of design, implementation, and optimization
- ✔ fast development
- ✘ staff must be trained to use the DSL
- ✘ interaction of DSL-generated software with other software components can be difficult
- ✘ there is high cost in developing and maintaining a DSL

Reference: J. Lawall and T. Mogensen. Course on Scripting Languages and DSLs, Univ. Copenhagen, 2006, `www.diku.dk/undervisning/2006f/213`

6. Software product lines

A software product line

is also called a *software system family* — a collection of software products that share an architecture and components, constructed by a product line. They are inspired by the products produced by industrial assembly lines, e.g., automobiles.

The CMU Software Engineering Institute definition:

A *product line* is a set of software intensive systems that
(i) share a common set of features,
(ii) satisfy the needs of a particular mission, and
(iii) are developed from a set of core assets in a prescribed way.

Key issues:
variability: Can we state precisely the products' variations (*features*) ?
guidance: Is there a precise recipe that guides feature selection and product assembly?

Reference: `www.softwareproductlines.com`

An example product line: Cummins Corporation

produces diesel engines for trucks and heavy machinery. An engine controller has 100K-200K lines-of-code. At level of 12 engine "builds," company switched to a product line approach:

1. defined engine controller domain
2. defined a reference architecture
3. built reusable components
4. required all teams to follow product line approach

Cummins now produces 20 basic "builds" — 1000 products total; development time dropped from 250 person/months to $<$ 10. A new controller consists of 75% reused software.

Reference: S. Cohen. Product line practice state of the art report. CMU/SEI-2002-TN-017.

Features and feature diagrams

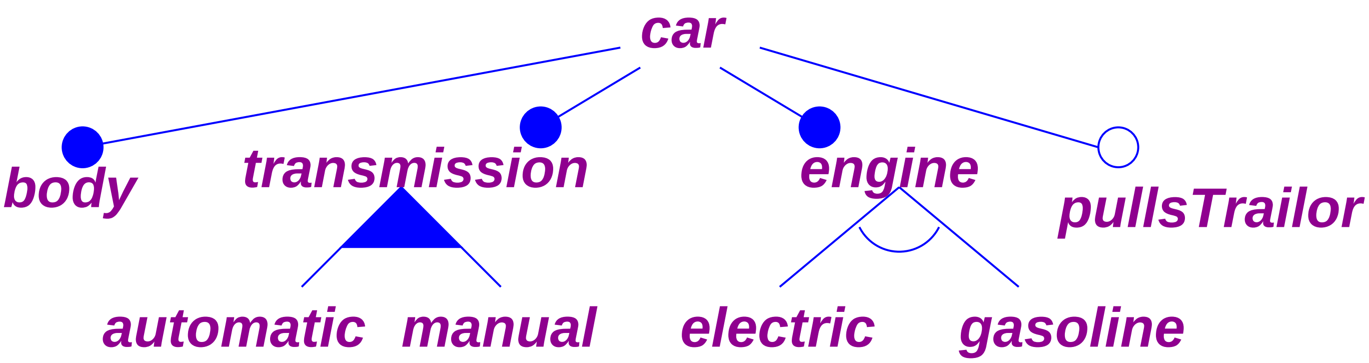
are a development tool for domain-specific architectures and product lines. They help define a domain's reference requirements and guide implementions of instances of the reference architecture.

A *feature* is merely a property of the domain. (Example: the features/options/choices of an automobile that you order from the factory.)

A *feature diagram* displays the features and guides a user in choosing features for the solution to a domain problem.

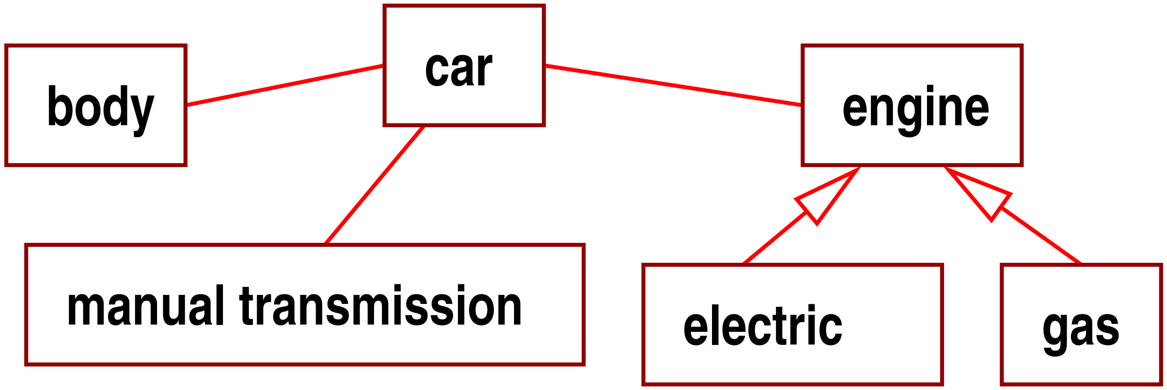
It is a form of decision tree with *and-or-xor* branching, and its hierarchy reflects dependencies of features as well as modification costs.

Feature diagram for assembling automobiles

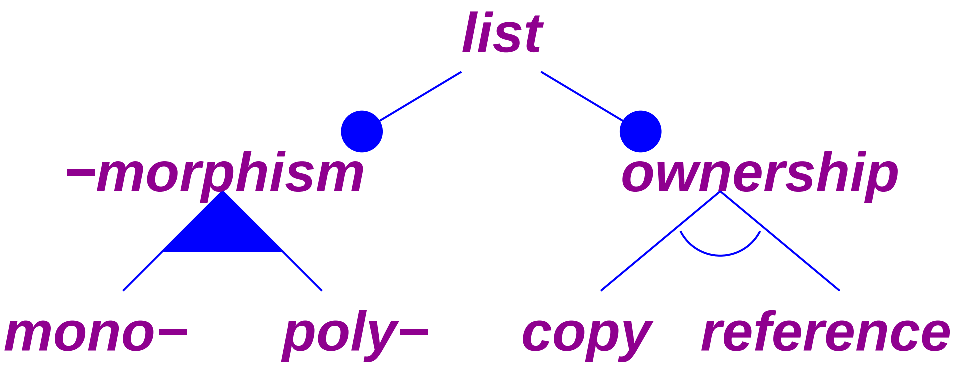


Filled circles label required features; unfilled circles label optional ones. Filled arcs label xor-choices; unfilled arcs label or-choices (where at least one choice is selected).

Here is one possible outcome of "executing" the feature diagram:



Feature diagrams work well for configuring generic data structures:



Compare the diagram to the typical class-library representation of a generic list structure.

An advantage of a feature-diagram construction of a list structure over a class-library construction is that the former can generate a smaller, more efficient list structure, customized to exactly the choices of the client.

Feature diagrams are useful for both *constraining* as well as *generating* an architecture: the feature requirements are displayed in a feature diagram, which guides the user to generating the desired instance of the reference architecture.

Feature diagrams are an attempt at making software assembly appear similar to assembly of mass-produced products like automobiles.

In particular, feature diagrams encourage the use of *standardized, parameterized, reusable software components*.

Feature diagrams might be implemented by a tool that selects components according to feature selection. Or, they might be implemented within the structure of a *domain-specific programming language* whose programs select and assemble features.

Reference: K. Czarnecki and U. Eisenecker. *Generative Programming.* Addison-Wesley 2000.

Generative programming

is the name given to the application of programs that generate other programs (cf. “automatic programming” in the 1950s). A compiler is of course a generating program, but so are feature-diagram-driven frameworks, partial evaluators, and some development environments (e.g., for Java beans).

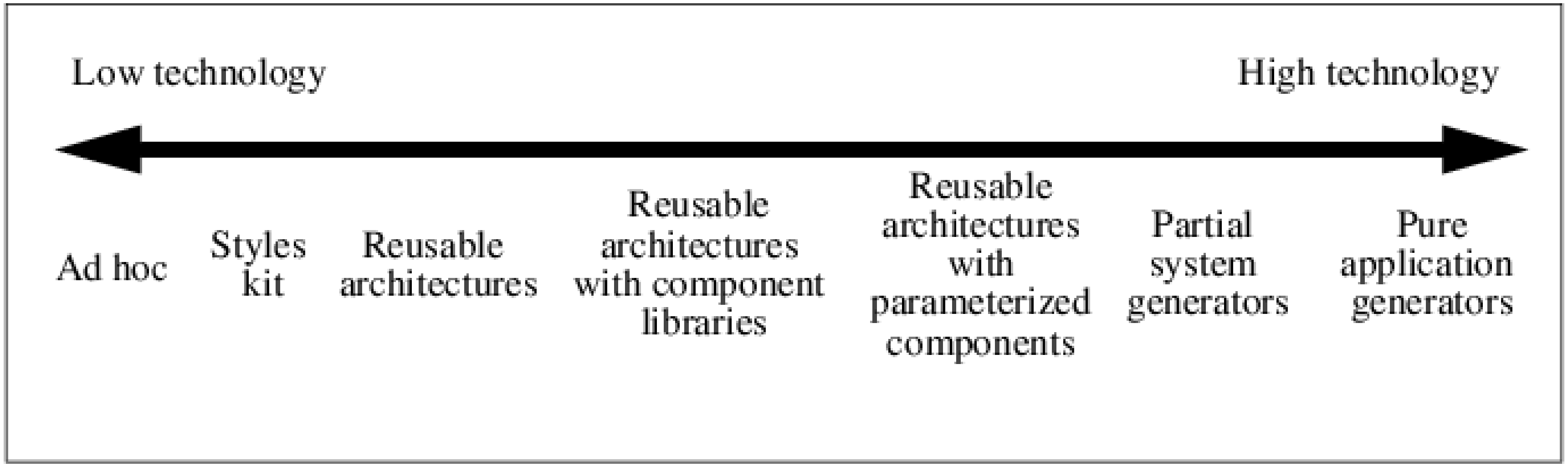
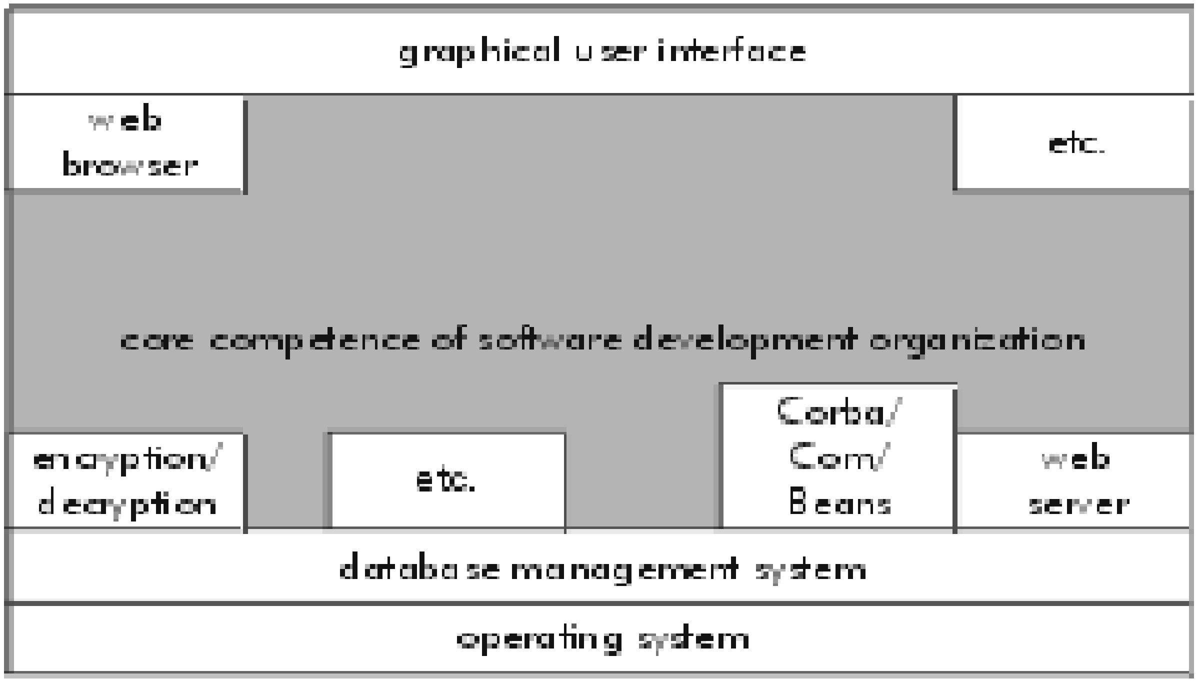

Low technology
High technology
Ad hoc
Styles kit
Reusable architectures
Reusable architectures with component libraries
Reusable architectures with parameterized components
Partial system generators
Pure application generators

Figure 1: Technology spectrum for architecture selection and creation

Reference: Coming attractions in software architecture, P. Clements. CMU/SEI-96-TR-008.

Generative programming is motivated by the belief that conventional software production methods (even those based on “object-oriented” methodologies) will never support component reuse:



software product

Reference: Jan Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 2000.

One solution is to understand a software system as a customized product, produced by generative programming, from a product line.

Reference: K. Czarnecki and U. Eisenecker. *Generative Programming.* Addison-Wesley 2000.

10. Final Remarks

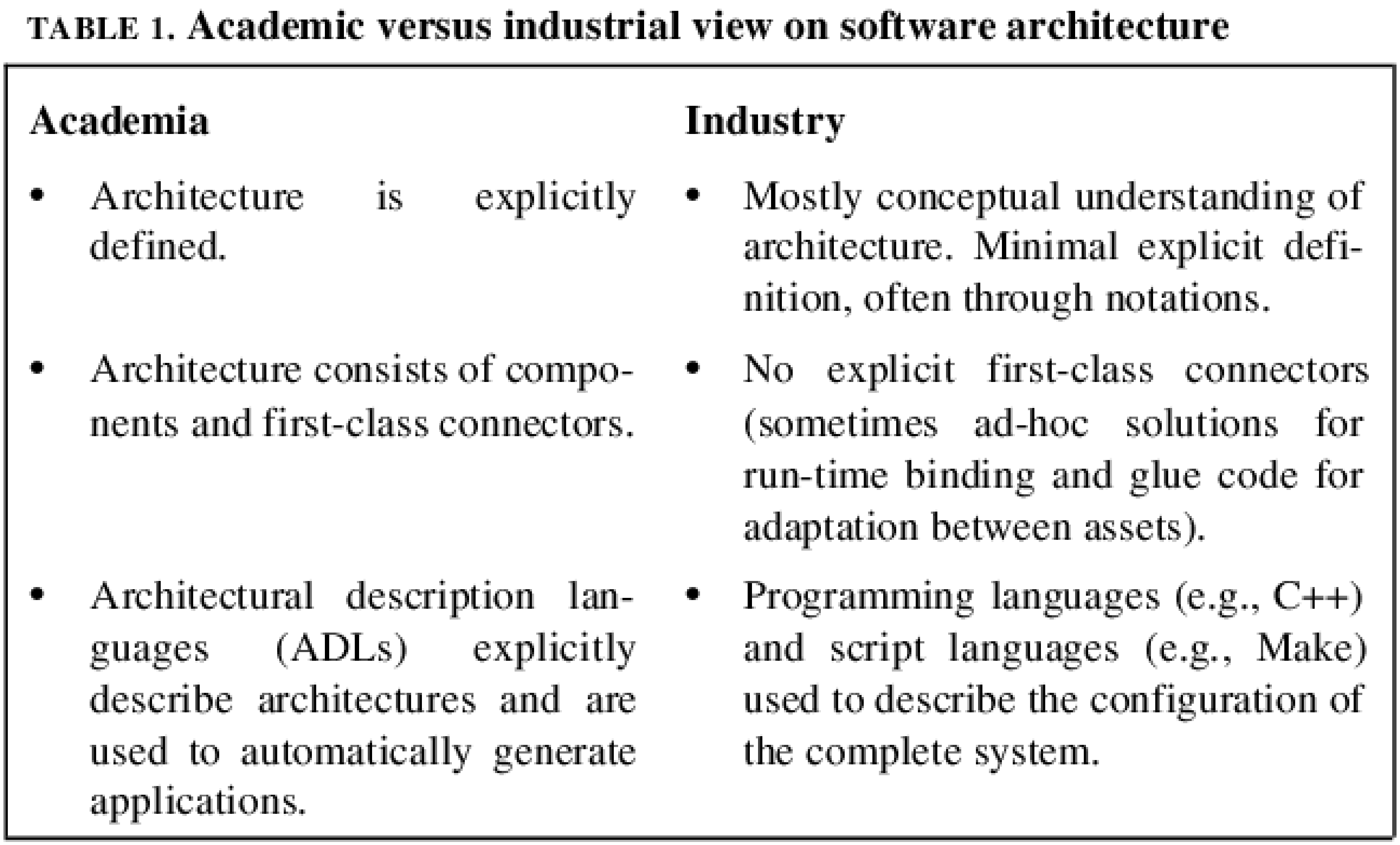


TABLE 1. **Academic versus industrial view on software architecture**

Academia	Industry
• Architecture is explicitly defined.	• Mostly conceptual understanding of architecture. Minimal explicit definition, often through notations.
• Architecture consists of components and first-class connectors.	• No explicit first-class connectors (sometimes ad-hoc solutions for run-time binding and glue code for adaptation between assets).
• Architectural description languages (ADLs) explicitly describe architectures and are used to automatically generate applications.	• Programming languages (e.g., C++) and script languages (e.g., Make) used to describe the configuration of the complete system.

Reference: Jan Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 2000.

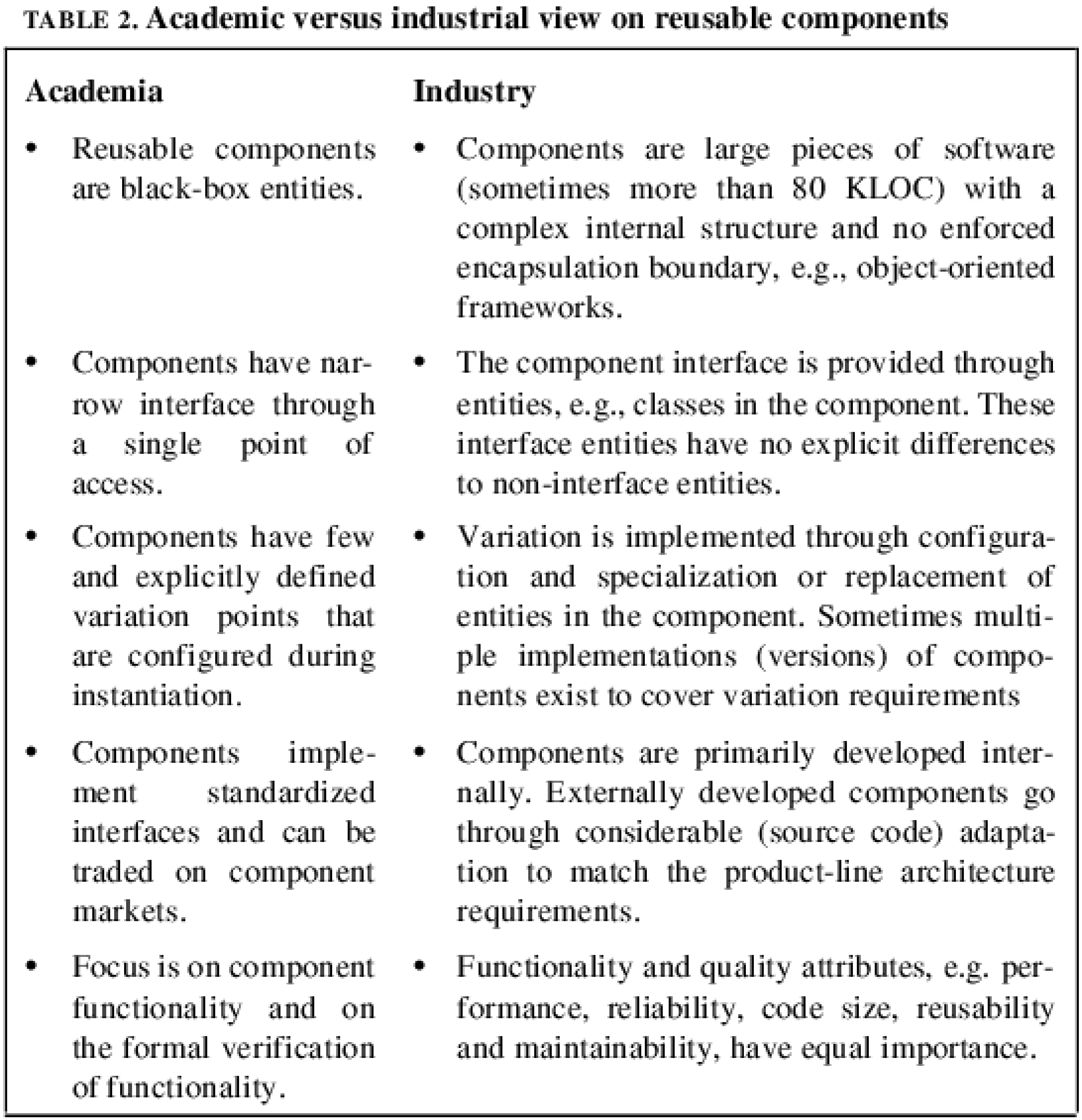


TABLE 2. **Academic versus industrial view on reusable components**

Academia	Industry
• Reusable components are black-box entities.	• Components are large pieces of software (sometimes more than 80 KLOC) with a complex internal structure and no enforced encapsulation boundary, e.g., object-oriented frameworks.
• Components have narrow interface through a single point of access.	• The component interface is provided through entities, e.g., classes in the component. These interface entities have no explicit differences to non-interface entities.
• Components have few and explicitly defined variation points that are configured during instantiation.	• Variation is implemented through configuration and specialization or replacement of entities in the component. Sometimes multiple implementations (versions) of components exist to cover variation requirements
• Components implement standardized interfaces and can be traded on component markets.	• Components are primarily developed internally. Externally developed components go through considerable (source code) adaptation to match the product-line architecture requirements.
• Focus is on component functionality and on the formal verification of functionality.	• Functionality and quality attributes, e.g. performance, reliability, code size, reusability and maintainability, have equal importance.

Selected textbook references

F. Buschmann, et al. *Pattern-Oriented Software Architecture.* Wiley 1996.

P. Clements and L. Northrup. *Software Product Lines*. Addison-Wesley 2002.

P. Clements, et al. *Documenting Software Architectures: Views and Beyond*. Addison Wesley, 2002.

K. Czarnecki and U. Eisenecker. *Generative Programming.* Addison-Wesley 2000.

E. Gamma, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

M. Shaw and D. Garlan. *Software Architecture.* Prentice Hall 1996.