KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore – 641 021.

(For the candidates admitted from 2017 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

SUBJECT : INTRODUCTION TO SOFTWARE ARCHITECTURE
SEMESTER : II
SUBJECT CODE : 17CSP204 **CLASS : I M.Sc.CS**

COURSE OBJECTIVE:

This course introduces basic concepts and principles about software design and software architecture. It starts with discussion on design issues followed by coverage on design patterns. It then gives an overview of architectural structures and styles. Practical approaches and methods for creating and analyzing software architecture are presented. The emphasis is on the interaction between quality attributes and software architecture. Students will also gain experiences with examples in design pattern application and case studies in software architecture.

COURSE OUTCOME:

A student who successfully completes this course should at the minimum be able to:

- Design and motivate software architecture for large scale software systems
- Recognize major software architectural styles design patterns and frameworks
- Generate architectural alternatives for a problem and select among them
- Use well-understood paradigms for designing new systems
- Identify and assess the quality attributes of a system at the architectural level

.

UNIT I

Introduction – Software Architecture – Software Design levels – An Engineering Discipline for Software – The status of Software Architecture – Architectural styles – Pipes and filters – Data Abstraction and Object-oriented organization – Event based implicit invocation – Layered systems – Repositories – Interpreters – Process Control – Other Familiar Architecture – Heterogeneous Architectures.

UNIT II

Case studies - Key word is Context – Instrumentation Software – Mobile Robotics – Cruise Control – Three Vignettes in Mixed Style

UNIT III

Shared Information Systems – Database Integration – Integration in Software Development Environments – Integration in the Design of Buildings – Architectural structures for shared Information Systems

UNIT IV

Guidance for User-Interface Architectures – The quantified Design Space – The value of Architectural formalism – Formalizing the Architecture of a specific system – Formalizing an Architectural Style – Formalizing an Architectural Design Space – Towards a Theory of Software Architecture – Z Notation

UNIT V

Requirements for Architecture – Description Languages – First class connectors – Adding Implicit Invocation to Traditional Programming Languages – Tools for Architectural Design – UniCon – Exploiting Style in Architectural Design Environments – Beyond definition/Use: Architectural Interconnection

SUGGESTED READINGS

TEXT BOOKS

1. Mary Shaw., & David Garlan. Software Architecture – Perspectives on an Emerging Discipline. New Delhi: Prentice Hall of India Eastern Economy edition.
2. Taylor Nenad., Medvidovic Eric., Dashofy, V., & Richard, N. (2010). Software Architecture: Foundations Theory and Practice. New Delhi: Wiley India Pvt. Limited.

REFERENCES

1. Boris Beizer. (1990). Software Testing Techniques (2nd ed.). Van Nostrand Reinhold.

ESE MARKS ALLOCATION

1.	**Section A** 20 x 1 = 20	20
2.	**Section B** 5 x 6 = 40 Either 'A' or 'B' choice	30
3.	**Section C** 1 x 10 = 10 Compulsory Question	10
	Total	60

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

Coimbatore – 641 021.

LECTURE PLAN
DEPARTMENT OF COMPUTER SCIENCE

STAFF NAME: N. MANONMANI

SUBJECT NAME: INTRODUCTION TO SOFTWARE ARCHITECTURE

SUB.CODE: 17CSP204

SEMESTER: II CLASS: I M.Sc (CS)

S.No.	Lecture Duration (Hours)	Topics to be Covered	Support Materials/Page Nos
		UNIT-I	
1.	1	Introduction: Software Architecture	T1: 1-3
2.	1	Software Design levels An Engineering Discipline for Software	T1: 4 T1: 5-14
3.	1	The status of Software Architecture	T1: 15-17
4.	1	Architectural styles	T1: 19-20
5.	1	Pipes and filters, Data Abstraction and Object-oriented organization	T1: 21-23
6.	1	Event based implicit invocation	T1: 23-24
7.	1	Layered systems	T1: 25-26
8.	1	Repositories, Interpreters	T1: 26, T1: 27
9.	1	Process Control	T1: 27-30
10.	1	Other Familiar Architecture, Heterogeneous Architectures	T1: 31 T1: 32
11.	1	Recapitulation and Discussion of important questions	
	Total No. of Hours Planned for Unit-I = 11		

		UNIT-II	
1.	1	Case studies - Key word is Context	T1: 33
2.	1	Solution 1: Main Program/Subroutine with Shared Data, Solution 2: Abstract Data Types	T1: 34-35
3.	1	Solution 3: Implicit Invocation, Solution 4: Pipes and Filters	T1:36-37
4.	1	Instrumentation Software: An Object-Oriented Model A Layered Model	T1: 39-40
5.	1	A Pipe-and-Filter Model A Modified Pipe-and-Filter	T1: 40-42
6.	1	Mobile Robotics - Design Consideration	T1: 43
7.	1	Solution 1: Control Loops Solution 2: Layered Architecture	T1: 44-46
8.	1	Solution 3 : Implicit Invocation Solution 4: Blackboard Architecture Comparisons	T1: 47-51
9.	1	Cruise Control	T1: 51-58
10.	1	Three Vignettes in Mixed Style	T1: 60-66
11.	1	Recapitulation and Discussion of important questions	
	Total No. of Hours Planned for Unit-II = 11		
		UNIT-III	
1.	1	Shared Information Systems	T1: 69
2.	1	Database Integration: Batch Sequential, Simple Repository	T1 : 70-74
3.	1	Database Integration: Virtual Repository, Hierarchical Layers, Evolution of Shared Information Systems in Business Data Processing	T1: 75-80
4.	1	Integration in Software Development Environments	T1: 82
5.	1	Batch Sequential, Transition from Batch Sequential to Repository	T1 : 83-84

6.	1	Repository, Hierarchical Layers, Evolution of Shared Information Systems in Business Data Processing	T1: 85-88
7.	1	Integration in the Design of Buildings	T1: 88
8.	1	Repository	T1: 89
9.	1	Intelligent Controls, Evolution of Shared Information Systems in Business Data Processing	T1: 90-91
10.	1	Architectural structures for shared Information Systems	T1: 93-94
11.	1	Recapitulation and Discussion of important questions	
	Total No. of Hours Planned for Unit-III = 11		
		UNIT-IV	
1.	1	Guidance for User-Interface Architectures: Design Spaces and Rules	T1: 97-99
2.	1	A design space for User-Interface Architectures	T1: 100-109
3.	1	Design Rules for User-Interface Architectures Applying the Design Space: An Example A Validation Experiment How the Design Space was prepared	T1: 110-114
4.	1	The quantified Design Space Overview, Background	T1: 116-119
5.	1	Quantified Design Space	T1: 120-127
6.	1	The value of Architectural formalism	T1: 129
7.	1	Formalizing the Architecture of a specific system	T1: 130-132
8.	1	Formalizing an Architectural Style	T1: 133-138
9.	1	Formalizing an Architectural Design Space	T1: 139-141
10.	1	Towards a Theory of Software Architecture	T1: 142
11.	1	Z Notation	T1: 143-146

12.	1	Recapitulation and Discussion of important questions	
	Total No. of Hours Planned for Unit-IV = 12		
		UNIT-V	
1.	1	Requirements for Architecture- Description Languages	T1: 147
2.	1	The Linguistic Character of Architectural Description Desiderata for Architecture-Description Languages	T1: 148-154
3.	1	Problems with Existing Languages	T1: 155
4.	1	**First class connectors:** Current Practice Problems with Current Practice A Fresh View of Software System Composition	T1: 160-165
5.	1	Adding Implicit Invocation to Traditional Programming Languages	T1: 172-181
6.	1	Tools for Architectural Design: UniCon	T1: 183-189
7.	1	Components and Connectors, Abstraction and Encapsulation	T1: 185-186
8.	1	Types and Type Checking, Accommodating Analysis Tools	T1: 187-189
9.	1	Exploiting Style in Architectural Design Environments	T1: 190-203
10.	1	Beyond definition/Use: Architectural Interconnection Implementation versus Interaction, Example	T1: 204-207
11.	1	The WRIGHT Model of Architectural Description Reasoning about Architectural Descriptions A Brief Explanation of our Use of CSP	T1: 208-212
12.	1	Recapitulation and Discussion of important questions	
13.	1	Recapitulation and Discussion of previous semester question papers	

14.	1	Recapitulation and Discussion of previous semester question papers	
15.	1	Recapitulation and Discussion of previous semester question papers	
		Total No. of Hours Planned for Unit-V = 15	
Total Planned Hours	60		

TEXT BOOKS

1. Mary Shaw., & David Garlan. Software Architecture – Perspectives on an Emerging Discipline. New Delhi: Prentice Hall of India Eastern Economy edition.

2. Taylor Nenad., Medvidovic Eric., Dashofy, V., & Richard, N. (2010). Software Architecture: Foundations Theory and Practice. New Delhi: Wiley India Pvt. Limited.

REFERENCES

1. Boris Beizer. (1990). Software Testing Techniques (2nd ed.). Van Nostrand Reinhold.

UNIT-I

SYLLABUS

Introduction – Software Architecture – Software Design levels – An Engineering Discipline for Software – The status of Software Architecture – Architectural styles – Pipes and filters – Data Abstraction and Object-oriented organization – Event based implicit invocation – Layered systems – Repositories – Interpreters – Process Control – Other Familiar Architecture – Heterogeneous Architectures.

INTRODUCTION TO SOFTWARE ARCHITECTURE

What is Software Architecture?

As the size and complexity of software systems increase, the design and specification of overall system structure become more significant issues than the choice of algorithms and data structures of computation.

Software architecture involves the description of elements from which systems are built, interactions among these elements, patterns that guide their composition, and constraints on these patterns.

A particular system is defined as a collection of components and interactions among those components. Such a system may be used as a (composite) element in a larger system. Architectures are represented abstractly as box-and-line diagrams.

Architectural descriptions serve as a skeleton around which system properties can be fleshed out. So they play a vital role in exposing the ability of a system to meet its major system requirements.

The architecture of a software system defines that system in terms of computational components and interactions among those components.

Components: clients & servers, databases, filters, and layers in a hierarchical system.

Interactions at this level of design can be simple and familiar. Ex: procedure calls, shared variable access.

But they can also be complex and semantically rich. Ex: client-server protocols, database accessing protocols, synchronous event multicast, and piped streams.

In addition to structure and topology the architecture shows the correspondence between the system requirements and elements of the constructed system.

Example:

Just as good programmers recognized useful data structures in the late 1960s, good software system designers now recognize useful system organizations.

One of these is based on the theory of abstract data types. But this is not the only way to organize a software system. Many other organizations have developed informally over time, and are now part of the vocabulary of software system designers. For example, typical descriptions of software architectures include synopses such as (italics ours):

•"Camelot is based on the *client-server model* and uses remote procedure calls both locally and remotely to provide communication among applications and servers."

•"*Abstraction layering* and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices.

The architecture encourages a *client server model* for the structuring of applications."

•"We have chosen a *distributed, object-oriented approach* to managing information."

•"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. . A more effective way [is to] split the source code into many segments, which are concurrently processed through the various phases of compilation [by multiple compiler processes] before a final, merging pass recombines the object code into a single program."

Other software architectures are carefully documented and often widely disseminated. Examples include the International Standard Organization's

Open Systems Interconnection Reference Model (a layered network architecture) , the NIST/ECMA Reference Model (a generic software engineering environment architecture based on layered communication substrates) , and the X Window System (a distributed windowed user interface architecture based on event triggering and callbacks) .

We are still far from having a well-accepted taxonomy of such architectural paradigms, let alone a fully-developed theory of software architecture. But we can now clearly identify a number of architectural patterns, or styles, that currently form the basic repertoire of a software architect.

As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.

This is the software architecture level of design. There is a considerable body of work on this topic, including module interconnection languages, templates and frameworks for systems that serve the needs of specific domains, and formal models of component integration mechanisms. In addition, an implicit body of work exists in the form of descriptive terms used informally to describe systems. And while there is not currently a well-defined terminology or notation to characterize architectural structures, good software engineers make common use of architectural principles when designing complex software. Many of the principles represent rules of thumb or idiomatic patterns that have emerged informally over time. Others are more carefully documented as industry and scientific standards.

It is increasingly clear that effective software engineering requires facility in architectural software design.

- First, it is important to be able to recognize common paradigms so that high-level relationships among systems can be understood and so that new systems can be built as variations on old systems.
- Second, getting the right architecture is often crucial to the success of a software system design; the wrong one can lead to disastrous results.
- Third, detailed understanding of software architectures allows the engineer to make principled choices among design alternatives.
- Fourth, an architectural system representation is often essential to the analysis and description of the high-level properties of a complex system.

SOFTWARE DESIGN LEVELS:

System design takes place at many levels. At each level we find

-components, both primitive and composite;

-rules of composition that allow the construction of non-primitive components or systems;

-rules of behavior that provide semantics for the system

There may be different notations, design problems, analysis techniques at each level.

Software, too, has its design levels.

1. Architecture:

- The design issues involve overall association of system capability with components.
- Components are modules, and inter connections among modules are handled in a variety of ways.
- Operators guide the composition of systems from subsystems.

2. Code:

- 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.

3. Executable:

- The design issues involve memory maps, data layouts, call stacks, and register allocations.
- The components are bit patterns supported by hardware.
- The operations and compositions are described in machine code.

Our concern here is to improve 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 connects the components; and system patterns often guide the compositions.

AN ENGINEERING DISCIPLINE FOR SOFTWARE:

What is Engineering?

Software engineering is a label applied to a set of current practices for software development. The more common usage refers to the disciplined application of scientific knowledge to resolve conflicting constraints and requirements for problems of immediate, practical significance.

Definitions of engineering share some common clauses

Creating cost-effective solutions…..

.... to practical problems

…. by applying scientific knowledge...

…. building things...

…. in the service of mankind

Routine and Innovative Design

Engineering design tasks are of several kinds; one of the most significant distinctions among them separates routine from innovative design.

Routine Design: involves solving familiar problems, reusing large portions of prior solutions.

Innovative Design: It involves finding novel solutions to unfamiliar problems. One path to increased productivity is identifying applications that could be routine and¬ developing appropriate support.

A Model for the Evolution of Engineering Discipline

Engineering has emerged from ad hoc practice in two stages.

1. Management and production techniques enable routine production.

2. The problems of routine production stimulate the development of a supporting science; the mature science eventually merges with established practice to yield professional engineering practice.

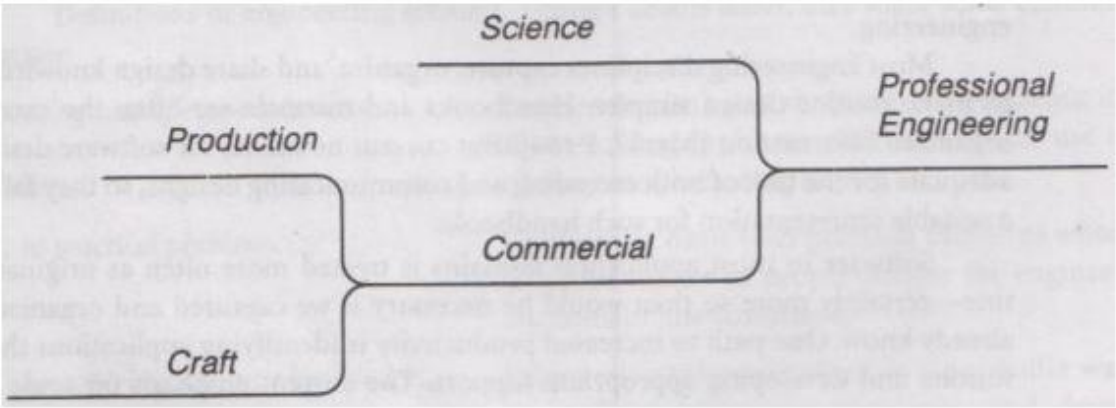


Fig: Evolution of an Engineering Discipline

The lower lines track the technology, and the upper lines show how the entry of production skills and scientific knowledge contribute new capability to the engineering practice.

The Current State of Software Technology

The engineering problem is creating cost-effective solutions to practical problems... building things in the service of mankind.

Scientific basis for Engineering practice

Engineering practice emerges from commercial practice by exploiting the results of a companion science.

One characterization of progress in programming languages and tools has been regular increases in abstraction level—or the conceptual size of software designers building blocks. To place the field of Software Architecture into perspective let us begin by looking at the historical development of abstraction techniques in computer science.

High-level Programming Languages

When digital computers emerged in the 1950s, software was written in machine language; programmers placed instructions and data individually and explicitly in the computer's memory. Insertion of a new instruction in a program might require hand-checking of the entire program to update references to data and instructions that moved as a result of the insertion. Eventually it was recognized that the memory layout and update of references could be automated, and also that symbolic names could be used for operation codes, and memory addresses. Symbolic assemblers were the result. They were soon followed by macro processors, which allowed a single symbol to stand for a commonly-used sequence of instructions. The substitution of simple symbols for machine operation codes, machine addresses yet to be defined, and sequences of instructions was perhaps the earliest form of abstraction in software.

In the latter part of the 1950s, it became clear that certain patterns of execution were commonly useful—indeed, they were so well understood that it was possible to create them automatically from a notation more like mathematics than machine language. The first of these patterns were for evaluation of arithmetic expressions, for procedure invocation, and for loops and conditional statements. These insights were captured in a series of early high-level languages, of which Fortran was the main survivor.

Higher-level languages allowed more sophisticated programs to be developed, and patterns in the use of data emerged. Whereas in Fortran data types served primarily as cues for selecting the proper machine instructions, data types in Algol and it successors serve to state the programmer’s intentions about how data should be used. The compilers for these languages could build on experience with Fortran and tackle more sophisticated compilation problems. Among other things, they checked adherence to these intentions, thereby providing incentives for the programmers to use the type mechanism.

Progress in language design continued with the introduction of modules to provide protection for related procedures and data structures, with the separation of a module’s specification from its implementation, and with the introduction of abstract data types.

Maturity of supporting science:

- Each program contains algorithms and data structures.
- Algorithms and data structures began to be abstracted from individual programs.

Research on abstract data types dealt with such issues as the following:

- Specifications (abstract models, algebraic axioms)

- Software structure (bundling representation with algorithms)

- Language issues (modules, scope, user-defined types)

- Information hiding (protecting integrity of information not in specification)

- Integrity constraints (invariants of data structures)

- Rules for composition (declarations)

Abstract Data Types

In the late 1960s, good programmers shared an intuition about software development: If you get the data structures right, the effort will make development of the rest of the program much easier. The abstract data type work of the 1970s can be viewed as a development effort that converted this intuition into a real theory. The conversion from an intuition to a theory involved understanding

• *the software structure* (which included a representation packaged with its primitive operators),

• *specifications* (mathematically expressed as abstract models or algebraic axioms),

• *language issues* (modules, scope, user-defined types),

• *integrity of the result* (invariants of data structures and protection from other manipulation),

• *rules for combining types* (declarations),

• *information hiding* (protection of properties not explicitly included in specifications).

The effect of this work was to raise the design level of certain elements of software systems, namely abstract data types, above the level of programming language statements or individual algorithms. This form of abstraction led to an understanding of a good organization for an entire module that serves one particular purpose. This involved combining representations, algorithms, specifications, and functional interfaces in uniform ways. Certain support was required from the programming language, of course, but the abstract data type paradigm allowed some parts of systems to be developed from a vocabulary of data types rather than from a vocabulary of programming-language constructs.

Interaction between Science and Engineering:

The development of good models within the software domain follows the pattern of following figure.

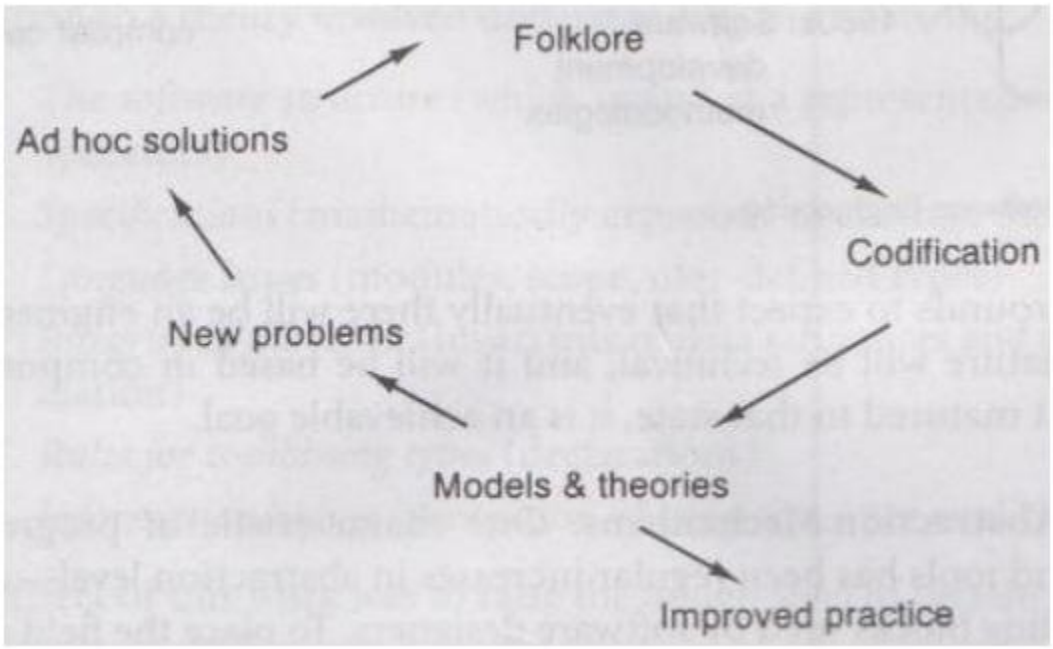


Fig: Codification Cycle for Science and Engineering

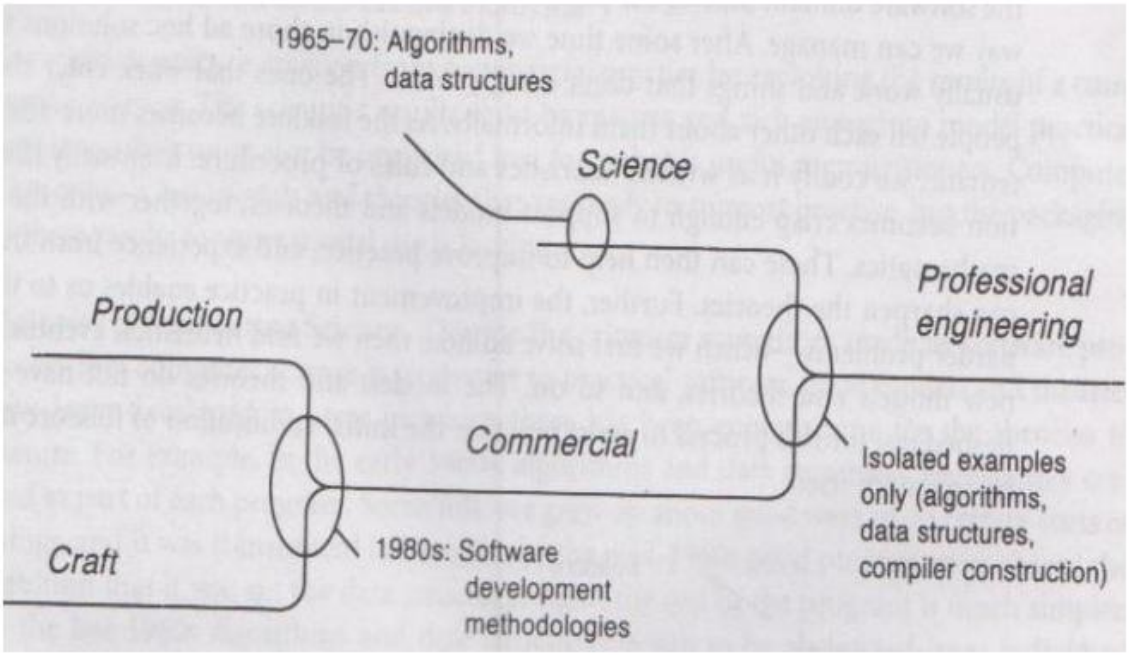


Fig: Evolution of Software Engineering

Codification through Abstraction Mechanisms:

One characteristic of progress in programming languages and tools has been regular increases in abstraction level—or the conceptual size of the building blocks used by software designers.

The conversion from an intuition to a theory involved understanding the following:

The software structure, Specifications, Language issues, Integrity of the result, Rules for combining types, Information hiding.

THE STATUS OF SOFTWARE ARCHITECTURE:

Good architectural design has always been a major factor in determining the success of a software system. Recently, software architecture has begun to emerge as an important field of study for software engineering practitioners and researchers.

Architectural issues are being addressed by work in areas such as module interface languages, domain-specific architectures, software reuse, codification of organizational patterns for software, architectural description languages and architectural design environments.

Two recent workshops brought together researchers and practitioners interested in software architecture to discuss the current state of the practice and art.

These workshops served to establish a common understanding of the state of the practice, the kinds of research and development efforts that are in progress, and the important challenges for this emerging field. Widespread interest in these workshops demonstrates the extent of these activities, which can be roughly placed into four categories.

- Addressing the problem of architectural characterization by providing new architectural description languages.
- Addressing the codification of the architectural expertise.
- Addressing frameworks for specific domains.
- Addressing formal underpinnings for architecture.

Software architecture provides benefits for both development and maintenance. For development, effective software engineers require facility in architectural software design.

- Able to recognize common paradigms
- Getting the right architecture
- Detailed understanding of software architecture
- Architectural system representation
- Fluency in the use of notations

Beyond the development stage, documenting a system's structure and properties has various advantages for maintenance. Much of the time spent on maintenance goes to

understanding the existing code. Retaining the designers intentions about system organization should help maintainers preserve the system' design integrity

ARCHITECTURAL STYLES:

An **architectural pattern** is a general, reusable solution to a commonly occurring problem in software architecture within a given context. The architectural patterns address various issues in software engineering, such as computer hardware performance limitations, high availability and minimization of a business risk. Some architectural patterns have been implemented within software frameworks.

An **architectural style** defines: a family of systems in terms of a pattern of structural organization; a vocabulary of components and connectors, with constraints on how they can be combined.

Some treat architectural patterns and architectural styles as the same, some treat styles as specializations of patterns.

The main difference is that a pattern can be seen as a solution to a problem, while a style is more general and does not require a problem to solve for its appearance. List of common architectural styles:

Dataflow systems: ▪ Batch sequential ▪ Pipes and filters **Call-and-return systems:** ▪ Main program and subroutine ▪ OO systems	**Virtual machines:** ▪ Interpreters ▪ Rule-based systems **Data-centered systems:** ▪ Databases ▪ Hypertext systems ▪ Blackboards

Hierarchical layers **Independent components:** ▪ Communicating processes ▪ Event systems	

PIPES AND FILTERS:

- Each components has set of inputs and set of outputs
- A component reads streams of data on its input and produces streams of data on its output.
- By applying local transformation to the input streams and computing incrementally, so that output begins before input is consumed. Hence, components are termed as filters.
- Connectors of this style serve as conducts for the streams transmitting outputs of one filter to inputs of another. Hence, connectors are termed pipes.

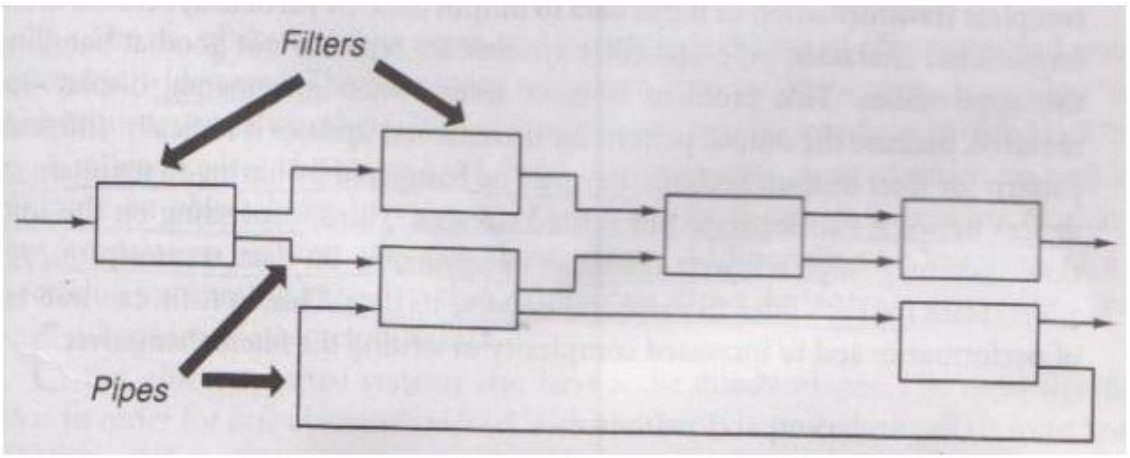


Fig : Pipes and Filters

Conditions (invariants) of this style are:

- Filters must be independent entities.

- They should not share state with other filter
- Filters do not know the identity of their upstream and downstream filters.
- Specification might restrict what appears on input pipes and the result that appears on the output pipes.
- Correctness of the output of a pipe-and-filter network should not depend on the order in which filter perform their processing.

Common specialization of this style includes:

- Pipelines: Restrict the topologies to linear sequences of filters.
- Bounded pipes: Restrict the amount of data that can reside on pipe.
- Typed pipes: Requires that the data passed between two filters have a well-defined type.

Batch sequential system:

A degenerate case of pipeline architecture occurs when each filter processes all of its input data as a single entity. In these systems pipes no longer serve the function of providing a stream of data and are largely vestigial.

Example 1: Best known examples of pipe-and-filter architecture are programs written in UNIXSHELL. Unix supports this style by providing a notation for connecting components [Unix process] and by providing run-time mechanisms for implementing pipes.

Example 2: Traditionally compilers have been viewed as pipeline systems. Stages in the pipeline include lexical analysis parsing, semantic analysis and code generation other examples of this type are.

- Signal processing domains
- Parallel processing
- Functional processing
- Distributed systems.

Advantages:

- They allow the designer to understand the overall input/output behavior of a system as a simple composition of the behavior of the individual filters
- They support reuse: Any two filters can be hooked together if they agree on data.
- Systems are easy to maintain and enhance: New filters can be added to exciting systems.
- They permit certain kinds of specialized analysis eg: deadlock, throughput
- They support concurrent execution.

Disadvantages:

- They lead to a batch organization of processing.
- Filters are independent even though they process data incrementally.
- Not good at handling interactive applications
    - When incremental display updates are required.
    - They may be hampered by having to maintain correspondences between two separate but related streams.
    - Lowest common denominator on data transmission.
- This can lead to both loss of performance and to increased complexity in writing the filters.

In a pipe and filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order. This is usually accomplished by applying a local transformation to the input streams and computing incrementally so output begins before input is consumed.

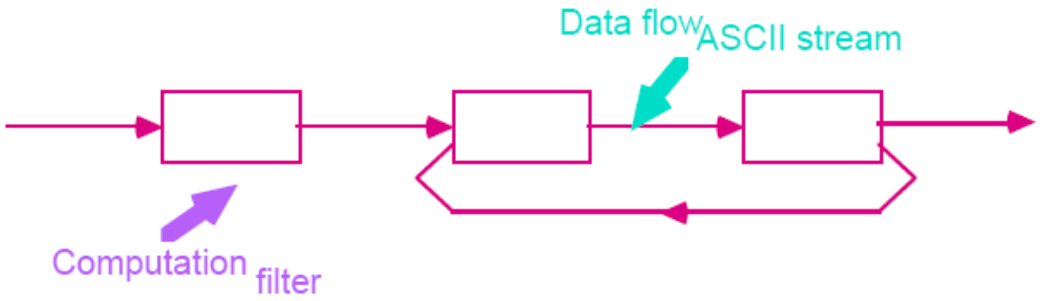
Hence components are termed “filters”. The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed “pipes”. Among the important invariants of the style, filters must be independent entities: in

particular, they should not share state with other filters. Another important invariant is that filters do not know the identity of their upstream and downstream filters.

Their specifications might restrict what appears on the input pipes or make guarantees about what appears on the output pipes, but they may not identify the components at the ends of those pipes.

Furthermore, the correctness of the output of a pipe and filter network should not depend on the order in which the filters perform their incremental processing—although fair scheduling can be assumed. (See [5] for an in-depth discussion of this style and its formal properties.) Figure 1 illustrates this style.

Common specializations of this style include *pipelines*, which restrict the topologies to linear sequences of filters; bounded pipes, which restrict the amount of data that can reside on a pipe; and typed pipes, which require that the data passed between two filters have a well-defined type.



A degenerate case of a pipeline architecture occurs when each filter processes all of its input data as a single entity.1 In this case the architecture becomes a “batch sequential” system. In these systems pipes no longer serve the function of providing a stream of data, and therefore are largely vestigial.

Hence such systems are best treated as instances of a separate architectural style.

The best known examples of pipe and filter architectures are programs written in the Unix shell. Unix supports this style by providing a notation for connecting components (represented as Unix

processes) and by providing run time mechanisms for implementing pipes. As another well-known example, traditionally compilers have been viewed as a pipeline systems (though the phases are often not incremental).

The stages in the pipeline include lexical analysis, parsing, semantic analysis, code generation. (We return to this example in the case studies.) Other examples of pipes and filters occur in signal processing domains, functional programming , and distributed systems.

Pipe and filter systems have a number of nice properties.

- First, they allow the designer to understand the overall input/output behavior of a system as a simple composition of the behaviors of the individual filters.
- Second, they support reuse: any two filters can be hooked together r, provided they agree on the data that is being transmitted between them.
- Third, systems can be easily maintained and enhanced: new filters can be added to existing systems and old filters can be replaced by improved ones.
- Fourth, they permit certain kinds of specialized analysis, such as throughput and deadlock analysis.
- Finally, they naturally support concurrent execution.

Each filter can be implemented as a separate task and potentially executed in parallel with other filters. But these systems also have their disadvantages.2 First, pipe and filter systems often lead to a batch organization of processing.

Although filters can process data incrementally, since filters are inherently independent, the designer is forced to think of each filter as providing a complete transformation of input data to output data. In particular, because of their transformational character, pipe and filter systems are typically not good at handling interactive applications.

- This problem is most severe when incremental display updates are required, because the output pattern for incremental updates is radically different from the pattern for filter output.

- Second, they may be hampered by having to maintain correspondences between two separate, but related streams.
- Third, depending on the implementation, they may force a lowest common denominator on data transmission, resulting in added work for each filter to parse and unparse its data.
- This, in turn, can lead both to loss of performance and to increased complexity in writing the filters themselves.

DATA ABSTRACTION AND OBJECT-ORIENTED ORGANIZATION:

In this approach, data representation and their associated primitive operations are encapsulated in the abstract data type (ADT) or object. The components of this style are objects/ADT‟s objects interact through function and procedure invocations. Objects are examples of a type of component we call a **manager** because it is responsible for preserving the integrity of a resource.

Two important aspects of this style are:

- Object is responsible for preserving the integrity of its representation.
- Representation is hidden from other objects.

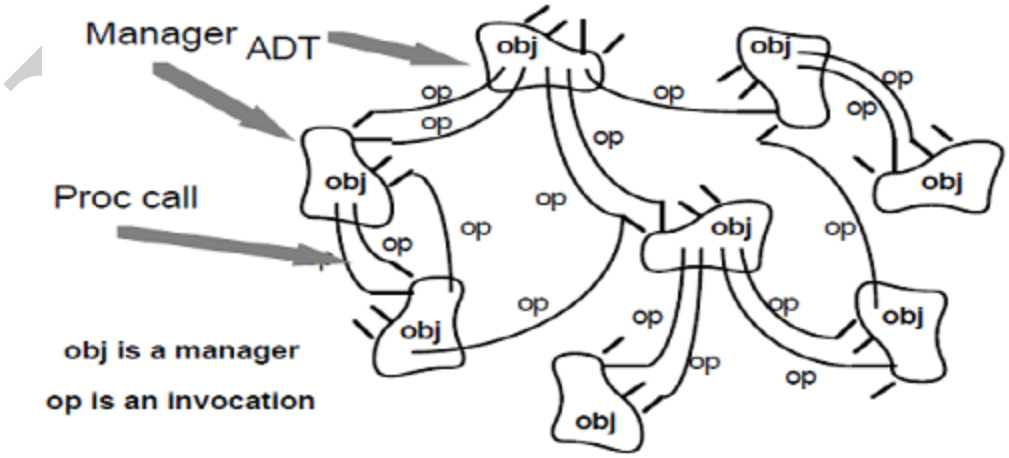


Fig: Abstract Data Types and Objects

Advantages:

- It is possible to change the implementation without affecting the clients because an object hides its representation from clients.
- The bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents.

Disadvantages:

- To call a procedure, it must know the identity of the other object.
- Whenever the identity of object changes it is necessary to modify all other objects that explicitly invoke it.

In this style data representations and their associated primitive operations are encapsulated in an abstract data type or object. The components of this style are the objects—or, if you will, instances of the abstract data types. Objects are examples of a sort of component we call a manager because it is responsible for preserving the integrity of a resource (here the representation). Objects interact through function and procedure invocations. Two important aspects of this style are (a) that an object is responsible for preserving the integrity of its representation (usually by maintaining some invariant over it), and (b) that the representation is hidden from other objects.

The use of abstract data types, and increasingly the use of object-oriented systems, is, of course, widespread. There are many variations. For example, some systems allow "objects" to be concurrent tasks; others allow objects to have multiple interfaces.

Object-oriented systems have many nice properties, most of which are well known. Because an object hides its representation from its clients, it is possible to change the implementation without affecting those clients. Additionally, the bundling of a set of accessing routines with the

data they manipulate allows designers to decompose problems into collections of interacting agents.

But object-oriented systems also have some disadvantages. The most significant is that in order for one object to interact with another (via procedure call) it must know the identity of that other object. This is in contrast, for example, to pipe and filter systems, where filters do need not know what other filters are in the system in order to interact with them. The significance of this is that whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it. In a module oriented language this manifests itself as the need to change the "import" list of every module that uses the changed module. Further there can be side effect problems: if A uses object B and C also uses B, then C's effects on B look like unexpected side effects to A, and vice versa.

EVENT-BASED, IMPLICIT INVOCATION:

- Instead of invoking the procedure directly a component can announce one or more events.
- Other components in the system can register an interest in an event by associating a procedure to it.
- When the event is announced, the system itself invokes all of the procedure that have been registered for the event. Thus an event announcement "implicitly" causes the invocation of procedures in other modules.
- Architecturally speaking, the components in an implicit invocation style are modules whose interface provides both a collection of procedures and a set of events.

Advantages:

- It provides strong support for reuse
- Any component can be introduced into the system simply by registering it for the events of that system.
- Implicit invocation eases system evolution.

- Components may be replaced by other components without affecting the interfaces of other components.

Disadvantages:

- Components relinquish control over the computation performed by the system.
- Concerns change of data
- Global performance and resource management can become artificial issues

Traditionally, in a system in which the component interfaces provide a collection of procedures and functions, components interact with each other by explicitly invoking those routines. However, recently there has been considerable interest in an alternative integration technique, variously referred to as implicit invocation, reactive integration, and selective broadcast. This style has historical roots in systems based on actors, constraint satisfaction, daemons, and packet-switched networks.

The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event. Thus an event announcement ``implicitly'' causes the invocation of procedures in other modules.

For example, in the Field system , tools such as editors and variable monitors register for a debugger's breakpoint events. When a debugger stops at a breakpoint, it announces an event that allows the system to automatically invoke methods in those registered tools. These methods might scroll an editor to the appropriate source line or redisplay the value of monitored variables. In this scheme, the debugger simply announces an event, but does not know what other tools (if any) are concerned with that event, or what they will do when that event is announced.

Architecturally speaking, the components in an implicit invocation style are modules whose interfaces provide both a collection of procedures (as with abstract data types) and a set of events. Procedures may be called in the usual way. But in addition, a component can register some of its procedures with events of the system. This will cause these procedures to be invoked when those events are announced at run time. Thus the connectors in an implicit invocation system include traditional procedure call as well as bindings between event announcements and procedure calls.

The main invariant of this style is that announcers of events do not know which components will be affected by those events. Thus components cannot make assumptions about order of processing, or even about what processing, will occur as a result of their events. For this reason, most implicit invocation systems also include explicit invocation (i.e., normal procedure call) as a complementary form of interaction.

Examples of systems with implicit invocation mechanisms abound.

They are used in programming environments to integrate tools, in database management systems to ensure consistency constraints, in user interfaces to separate presentation of data from applications that manage the data, and by syntax-directed editors to support incremental semantic checking.

One important benefit of implicit invocation is that it provides strong support for reuse. Any component can be introduced into a system simply by registering it for the events of that system. A second benefit is that implicit invocation eases system evolution. Components may be replaced by other components without affecting the interfaces of other components in the system.

In contrast, in a system based on explicit invocation, whenever the identity of a that provides some system function is changed, all other modules that import that module must also be changed.

The primary disadvantage of implicit invocation is that components relinquish control over the computation performed by the system. When a component announces an event, it has no idea what other components will respond to it. Worse, even if it does know what other components are interested in the events it announces, it cannot rely on the order in which they are invoked. Nor can it know when they are finished.

Another problem concerns exchange of data. Sometimes data can be passed with the event. But in other situations event systems must rely on a shared repository for interaction. In these cases global performance and resource management can become a serious issue. Finally, reasoning about correctness can be problematic, since the meaning of a procedure that announces events will depend on the context of bindings in which it is invoked. This is in contrast to traditional reasoning about procedure calls, which need only consider a procedure's pre- and post-conditions when reasoning about an invocation of it.

LAYERED SYSTEMS:

- A layered system is organized hierarchically, each layer provides service to the layer above it and serving as a client to the layer below.
- Inner layers are hidden from all except the adjacent layers.
- Connectors are defined by the protocols that determine how layers interact with each other.
- Goal is to achieve qualities of modifiability portability.

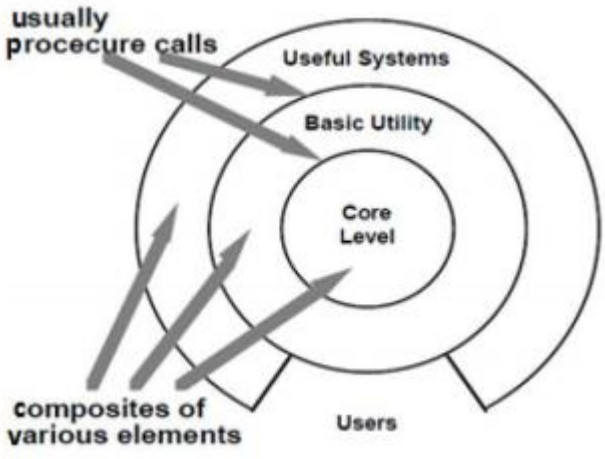


Fig: Layered Systems

Examples:

- Layered communication protocol
- Operating systems

Database systems

Advantages:

- They support designs based on increasing levels abstraction.
- Allows implementers to partition a complex problem into a sequence of incremental steps.
- They support enhancement
- They support reuse.

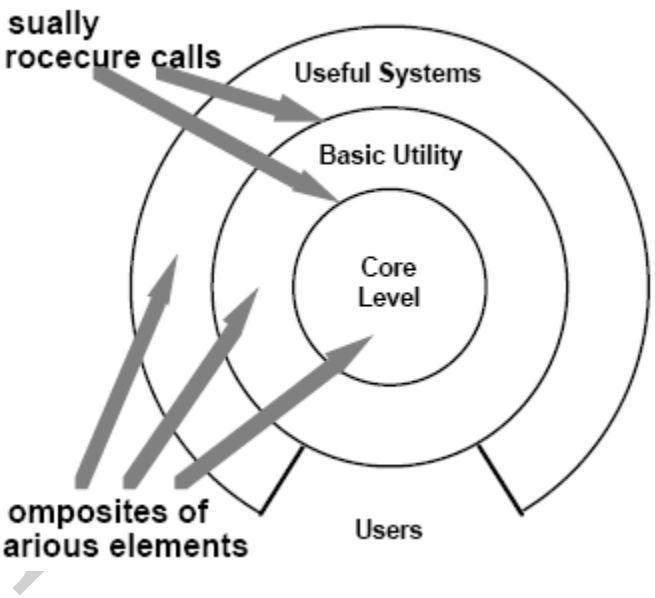
Disadvantages:

- Not easily all systems can be structures in a layered fashion.
- Performance may require closer coupling between logically high-level functions and their lower-level implementations.

- Difficulty to mapping existing protocols into the ISO framework as many of those
- protocols bridge several layers.

Layer bridging: functions in one layer may talk to other than its immediate neighbor.

A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. In some layered systems inner layers are hidden from all except the adjacent outer layer, except for certain functions carefully selected for export. Thus in these systems the components implement a virtual machine at some layer in the hierarchy. (In other layered systems the layers may be only partially opaque.) The connectors are defined by the protocols that determine how the layers will interact. Topological constraints include limiting interactions to adjacent layers. Figure illustrates this style.



The most widely known examples of this kind of architectural style are layered communication protocols. In this application area each layer provides a substrate for communication at some level of abstraction. Lower levels define lower levels of interaction, the lowest typically being

defined by hardware connections. Other application areas for this style include database systems and operating systems.

Layered systems have several desirable properties.

- First, they support design based on increasing levels of abstraction. This allows implementers to partition a complex problem into a sequence of incremental steps.
- Second, they support enhancement. Like pipelines, because each layer interacts with at most the layers below and above, changes to the function of one layer affect at most two other layers.
- Third, they support reuse. Like abstract data types, different implementations of the same layer can be used interchangeably, provided they support the same interfaces to their adjacent layers.
- This leads to the possibility of defining standard layer interfaces to which different implementers can build. (A good example is the OSI ISO model and some of the X Window System protocols.)

But layered systems also have disadvantages. Not all systems are easily structured in a layered fashion. And even if a system *can* logically be structured as layers, considerations of performance may require closer coupling between logically high-level functions and their lower-level implementations. Additionally, it can be quite difficult to find the right levels of abstraction. This is particularly true for standardized layered models. One notes that the communications community has had some difficulty mapping existing protocols into the ISO framework: many of those protocols bridge several layers.

In one sense this is similar to the benefits of implementation hiding found in abstract data types. However, here there are multiple levels of abstraction and implementation. They are also similar to pipelines, in that components communicate at most with one other component on either side.

But instead of simple pipe read/write protocol of pipes, layered systems can provide much richer forms of interaction. This makes it difficult to define system independent layers (as with filters)—since a layer must support the specific protocols at its upper and lower boundaries. But it also allows much closer interaction between layers, and permits two-way transmission of information.

REPOSITORIES: [data cantered architecture]

- Goal of achieving the quality of integrity of data.
- In this style, there are two kinds of components.

i. Central data structure- represents current state.

ii. Collection of independent components which operate on central data store.

The choice of a control discipline leads to two major sub categories.

- Type of transactions is an input stream trigger selection of process to execute
- Current state of the central data structure is the main trigger for selecting processes to execute. The Active repository can be a blackboard.

Blackboard:

Three major parts:

- **Knowledge sources:** Separate, independent parcels of application dependents knowledge.
- **Blackboard data structure:** Problem solving state data, organized into an application dependent hierarchy
- **Control:** Driven entirely by the state of blackboard

Invocation of a knowledge source (ks) is triggered by the state of blackboard.

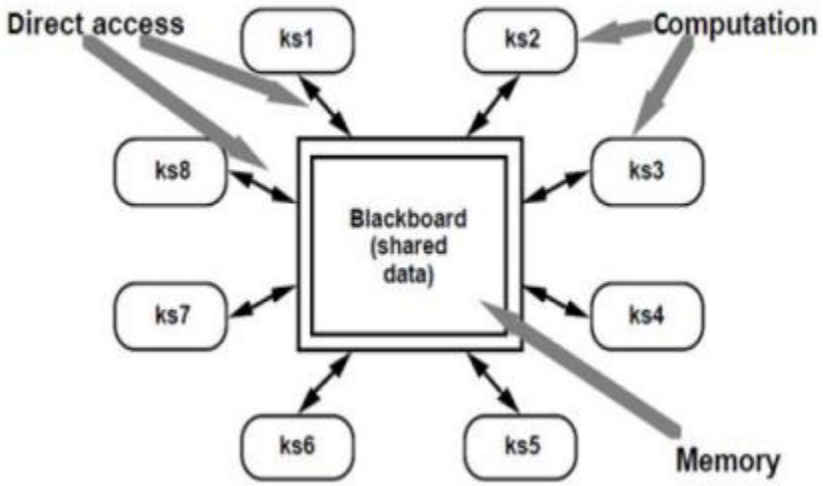


Fig: The Blackboard

The actual focus of control can be in

- Knowledge source
- Blackboard
- Separate module or
- Combination of these

Blackboard systems have traditionally been used for application requiring complex interpretation of signal processing like speech recognition, pattern recognition.

In a repository style there are two quite distinct kinds of components: a central data structure represents the current state, and a collection of independent components operate on the central data store. Interactions between the repository and its external components can vary significantly between systems.

The choice of control discipline leads to major subcategories. If the types of transactions in an input stream of transactions trigger selection of processes to execute, the repository can be a

traditional database. If the current state of the central data structure is the main trigger of selecting processes to execute, the repository can be a blackboard.

Figure illustrates a simple view of blackboard architecture. The blackboard model is usually presented with three major parts:

The knowledge sources: separate, independent parcels of application dependent knowledge. Interaction among knowledge sources takes place solely through the blackboard.

The blackboard data structure: problem-solving state data, organized into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.

Control: driven entirely by state of blackboard. Knowledge sources respond opportunistically when changes in the blackboard make them applicable.

In the diagram there is no explicit representation of the control component. Invocation of a knowledge source is triggered by the state of the blackboard. The actual locus of control, and hence its implementation, can be in the knowledge sources, the blackboard, a separate module, or some combination of these.

Blackboard systems have traditionally been used for applications requiring complex interpretations of signal processing, such as speech and pattern recognition. They have also appeared in other kinds of systems that involve shared access to data with loosely coupled agents.

There are, of course, many other examples of repository systems. Batch sequential systems with global databases are a special case. Programming environments are often organized as a collection of tools together with a shared repository of programs and program fragments.

Even applications that have been traditionally viewed as pipeline architectures may be more accurately interpreted as repository systems.

For example, as we will see later, while compiler architecture has traditionally been presented as a pipeline, the "phases" of most modern compilers operate on a base of shared information (symbol tables, abstract syntax tree, etc.).

INTERPRETERS:

- An interpreter includes pseudo program being interpreted and interpretation engine.
- Pseudo program includes the program and activation record.
- Interpretation engine includes both definition of interpreter and current state of its execution.

Interpreter includes 4 components:

1. Interpretation engine: to do the work
2. Memory: that contains pseudo code to be interpreted.
3. Representation of control state of interpretation engine
4. Representation of control state of the program being simulated.

Ex: JVM or "virtual Pascal machine"

Advantages:

Executing program via interpreters adds flexibility through the ability to interrupt and query the program.

Disadvantages: Performance cost because of additional computational involved.

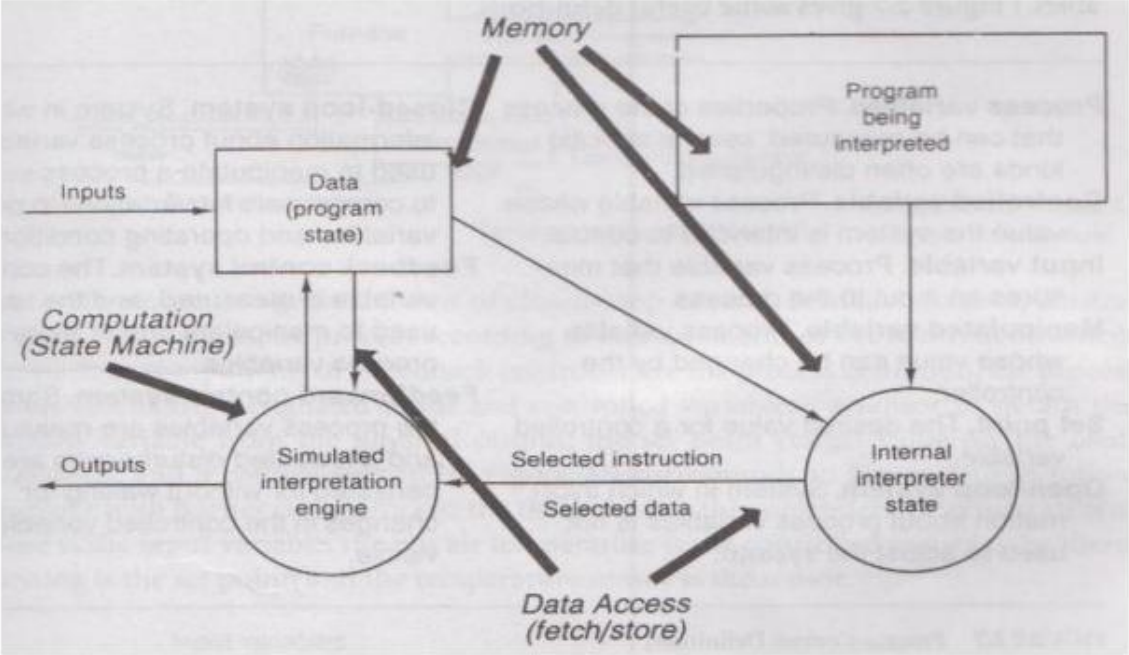


Fig: Interpreter

Table Driven Interpreters

In an interpreter organization a virtual machine is produced in software. An interpreter includes the pseudo-program being interpreted and the interpretation engine itself. The pseudo-program includes the program itself and the interpreter's analog of its execution state (activation record). The interpretation engine includes both the definition of the interpreter and the current state of *its* execution. Thus an interpreter generally has four components: an interpretation engine to do the work, a memory that contains the pseudo-code to be interpreted, a representation of the control state of the interpretation engine, and a representation of the current state of the program being simulated.

Interpreters are commonly used to build virtual machines that close the gap between the computing engine expected by the semantics of the program and the computing engine available in hardware. We occasionally speak of a programming language as providing, say, a "virtual Pascal machine." We will return to interpreters in more detail in the case studies.

PROCESS CONTROL:

This architectural style is based on control loops.

- Object-oriented and functional designs are characterized by the kinds of components that appear.
- Control loop designs are characterized by both the kinds of components involved and the special relations that must hold among them.

Process Control Paradigms:

Continuous processes of many kinds convert input materials to products with specific properties by performing operations on the inputs and on intermediate products.

Process Control Definitions:

Process variables: properties of the process that can be measured

Controlled variable: process variable whose value of the system is intended to control

Input variable: process variable that measures an input to the process

Manipulated variable: process variable whose value can be changed by the controller.

Set point: the desired value for a controlled variable

Open-loop system: system in which information about process variables is not used to adjust the system.

Closed-loop system: system in which information about process variables is used to manipulate a process variable to compensate for variations in process variables and operating conditions.

Feedback control system: the controlled variable is measured and the result is used to manipulate one or more of the process variables

Feed forward control system: some of the process variables are measured, and anticipated disturbances are compensated without waiting for changes in the controlled variable to be visible.

The open-loop assumptions are rarely valid for physical processes in the real world. More often, properties such as temperature, pressure and flow rates are monitored, and their values are used to control the process by changing the settings of apparatus such as valve, heaters and chillers. Such systems are called closed loop systems.

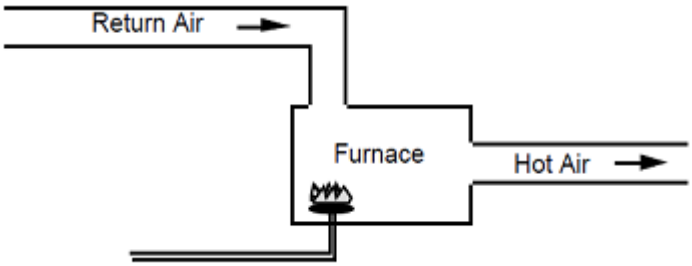


Fig: open-loop temperature control

A home thermostat is a common example; the air temperature at the thermostat is measured, and the furnace is turned on and off as necessary to maintain the desired temperature. Below figure shows the addition of a thermostat to convert above figure to a closed loop system.

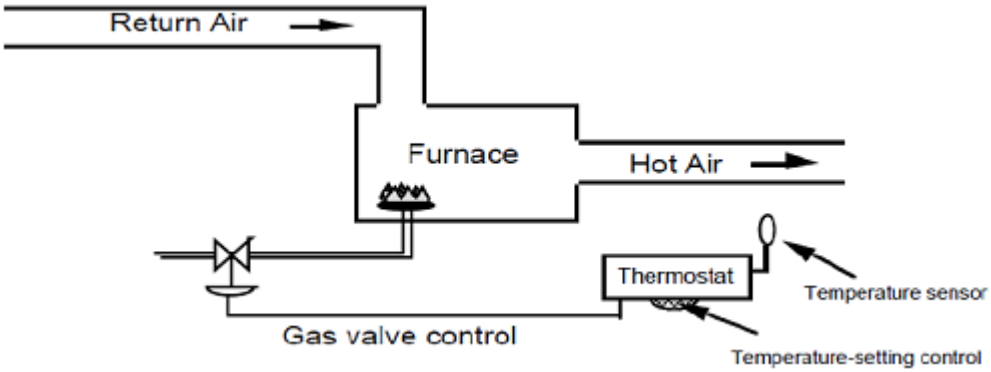


Fig: closed-loop temperature control

Feedback control:

Above figure corresponds to below figure as follows:

- The furnace with burner is the process
- The thermostat is the controller
- The return air temperature is the input variable
- The hot air temperature is the controlled variable
- The thermostat setting is the set point
- Temperature sensor is the sensor

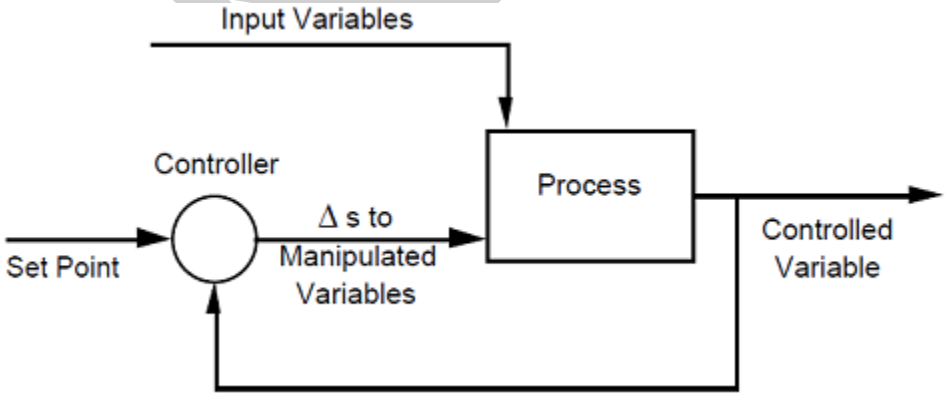


Fig: feedback control

Feedforward control:

It anticipates future effects on the controlled variable by measuring other process variables and adjusts the process based on these variables. The important components of a feed forward controller are essentially the same as for a feedback controller except that the sensor(s) obtain values of input or intermediate variables.

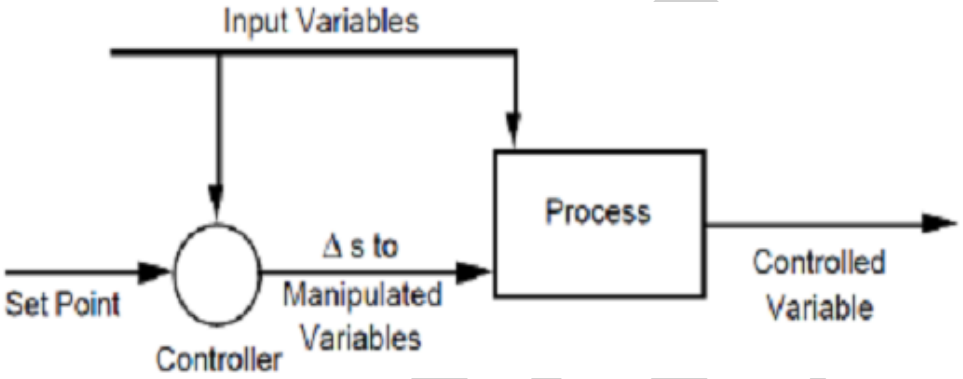


Fig: feed forward control

These are simplified models

- They do not deal with complexities - properties of sensors, transmission delays &
- calibration issues
- They ignore the response characteristics of the system, such as gain, lag and hysteresis.
- They don't show how combined feed forward and feedback
- They don't show how to manipulate process variables.

A Software Paradigm for Process Control:

An architectural style for software that controls continuous processes can be based on the process-control model, incorporating the essential parts of a process-control loop:

Computational elements: separate the process of interest from the controlled policy

- **Process definition,** including mechanisms for manipulating some process variables
- **Control algorithm,** for deciding how to manipulate variables

Data element: continuously updated process variables and sensors that collect them

- **Process variables,** including designed input, controlled and manipulated variables and knowledge of which can be sensed
- **Set point,** or reference value for controlled variable
- **Sensors** to obtain values of process variables pertinent to control

The control loop paradigm: establishes the relation that the control algorithm exercises.

OTHER FAMILIAR ARCHITECTURES:

- Distributed processes: Distributed systems have developed a number of common organizations for multi-process systems. Some can be characterized primarily by their topological features, such as ring and star organizations. Others are better characterized in terms of the kinds of inter-process protocols that are used for communication (e.g., heartbeat algorithms).
- Main program/subroutine organizations: The primary organization of many systems mirrors the programming language in which the system is written. For languages without support for modularization this often results in a system organized around a main program and a set of subroutines.
- Domain-specific software architectures: These architectures provide an organizational structure tailored to a family of applications, such as avionics, command and control, or vehicle management systems. By specializing the architecture to the domain, it is possible to increase the descriptive power of structures.
- State transition systems: These systems are defined in terms a set of states and a set of named transitions that move a system from one state to another.

There are numerous other architectural styles and patterns. Some are widespread and others are specific to particular domains. While a complete treatment of these is beyond the scope of this paper, we briefly note a few of the important categories.

• **Distributed processes:** Distributed systems have developed a number of common organizations for multi-process systems [37]. Some can be characterized primarily by their topological features, such as ring and star organizations. Others are better characterized in terms of the kinds of inter-process protocols that are used for communication (e.g., heartbeat algorithms).

One common form of distributed system architecture is a "client-server" organization [38]. In these systems a server represents a process that provides services to other processes (the clients). Usually the server does not know in advance the identities or number of clients that will access it at run time. On the other hand, clients know the identity of a server (or can find it out through some other server) and access it by remote procedure call.

• **Main program/subroutine organizations:** The primary organization of many systems mirrors the programming language in which the system is written. For languages without support for modularization this often results in a system organized around a main program and a set of subroutines. The main program acts as the driver for the subroutines, typically providing a control loop for sequencing through the subroutines in some order.

• **Domain-specific software architectures:** Recently there has been considerable interest in developing "reference" architectures for specific domains. These architectures provide an organizational structure tailored to a family of applications, such as avionics, command and control, or vehicle management systems. By specializing the architecture to the domain, it is possible to increase the descriptive power of structures. Indeed, in many cases the architecture is sufficiently constrained that an executable system can be generated automatically or semi-automatically from the architectural description itself.

• **State transition systems:** A common organization for many reactive systems is the state transition system [40]. These systems are defined in terms a set of states and a set of named transitions that move a system from one state to another.

• **Process control systems:** Systems intended to provide dynamic control of a physical environment are often organized as process control systems. These systems are roughly characterized as a feedback loop in which inputs from sensors are used by the process control system to determine a set of outputs that will produce a new state of the environment.

HETEROGENEOUS ARCHITECTURES:

Architectural styles can be combined in several ways:

- One way is through hierarchy. Example: UNIX pipeline
- Second way is to combine styles is to permit a single component to use a mixture of architectural connectors. Example: "active database"
- Third way is to combine styles is to completely elaborate one level of architectural description in a completely different architectural style. Example: case studies.

Thus far we have been speaking primarily of "pure" architectural styles.

- While it is important to understand the individual nature of each of these styles, most systems typically involve some combination of several styles. There are different ways in which architectural styles can be combined. One way is through hierarchy. A component of a system organized in one architectural style may have an internal structure that is developed a completely different style. For example, in a Unix pipeline the individual components may be represented internally using virtually any style— including, of course, another pipe and filter, system.
- What is perhaps more surprising is that connectors, too, can often be hierarchically decomposed. For example, a pipe connector may be implemented internally as a FIFO queue accessed by insert and remove operations.
- A second way for styles to be combined is to permit a single component to use a mixture of architectural connectors. For example, a component might access a repository through part of its interface, but interact through pipes with other components in a system, and

accept control information. (In fact, Unix pipe and filter systems do this, the file system playing the role of the repository and initialization switches playing the role of control.)

- Another example is an "active database". This is a repository which activates external components through implicit invocation. In this organization external components register interest in portions of the database. The database automatically invokes the appropriate tools based on this association. (Blackboards are often constructed this way; knowledge sources are associated with specific kinds of data, and are activated whenever that kind of data is modified.)
- A third way for styles to be combined is to completely elaborate one level of architectural description in a completely different architectural style.

POSSIBLE QUESTIONS

PART – B

1. Elucidate about An Engineering Discipline for Software.
2. Explain about Pipes and Filters and Object-oriented organization with example.
3. What is a Software Architecture? Discuss on Software Design Levels.
4. Discuss on Repositories and Interpreter.
5. Elaborate a model for Evolution of an Engineering Discipline with a Diagram.
6. Discuss in detail about Layered Systems and Repository style.
7. Elucidate on the Current state of Software Technology.
8. What are Architecture Styles? Explain.

PART – C

1. Compare and contrast the various types of Architectural Styles
2. Describe An Engineering Discipline for Software

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS: I MSC CS SUBJECT NAME: INTRODUCTION TO SOFTWARE ARCHITECTURE
SUBJECT CODE: 17CSP204 BATCH-2017-2019

UNIT: II (ONE MARKS) PART A - ONLINE EXAMINATION

S.NO	QUESTION	CHOICE 1	CHOICE 2	CHOICE 3	CHOICE 4	ANSWER
1	The KWIC means	Key Word in Context	Key word in Constant	Key word as Context	Key word as Capital	Key Word in Context
2	The KWIC index system outputs a listing of all _____ of all lines in alphabetical order	circular shifts	ordered set of words	ordered set of characters	set of lines	circular shifts
3	The _____ solution decomposes the problem according to the four basic functions performed: input, shift, alphabetize, and output.	Main program/Subroutine with shared data	abstract data type	layered	event based	Main program/Subroutine with shared data
4	Data is communicated between the components through _____.	shared storage	module	algorithm	data	shared storage
5	Communication between the computational components and the shared data is an unconstrained _____ protocol	read write	network	Communication	read	read write
6	In Abstract Data Types each module provides an interface that permits other components to access data only by invoking _____	shared storage	module	procedures in that interface	data	procedures in that interface
7	In _____ solution computations are invoked implicitly as data is modified.	Implicit Invocation	abstract data type	layered	process control	Implicit Invocation
8	The KWIC index system outputs a listing of all circular shifts of all lines in _____ order.	forward	alphabetical	backward	circle	alphabetical
9	In pipe and filter new functions are easily added to the system by _____ at the appropriate point in the processing sequence	inserting filter	editing filter	deleting filter	copying filter	inserting filter
10	The permuted" [sic] index for the Unix Man pages is an example of _____	KWIC	implicit invocation	instrumentation software	robotics	KWIC
11	The data types used in oscilloscopes is _____	waveforms, signals	int	float	char	waveforms, signals

12	_____ company work towards the software to support their Instrumentation products like oscilloscope	Dell laboratories	Computer Research Laboratory	Tektronix	Cipla	Tektronix
13	An oscilloscope is an instrumentation system that samples electrical signals and displays pictures_____ of them on a screen	traces	pulses	waves	lines	traces
14	In layered model the _____ represented the signal manipulation functions that filter signals as they enter the oscilloscope.	individual modules	core layer	outer layer	inner layer	core layer
15	In _____ layer signals are digitized and stored internally for later processing	waveform acquisition	hardware	user interface	visual	waveform acquisition
16	The outermost layer in oscilloscope's layered model is _____	user interface	waveform acquisition	hardware	visual	user interface
17	_____serve to condition external signals in pipe filter model of oscilloscope	Signal transformers	acquisition transformers	Display transformers	waveform transformers	Signal transformers
18	_____ derive digitized waveforms from these signals in pipe filter model of oscilloscope	Signal transformers	acquisition transformers	Display transformers	waveform transformers	acquisition transformers
19	_____convert these waveforms into visual data in pipe filter model of oscilloscope	Signal transformers	acquisition transformers	Display transformers	waveform transformers	Display transformers
20	The _____ solution accounted for user inputs by associating with each filter a control interface that allows an external entity to set parameters of operation for the filter	A Modified Pipe and Filter Model	robotics	implicit invocation	instrumentation software	A Modified Pipe and Filter Model
21	In _____ controls manned, partially-manned, or unmanned vehicle	Mobile Robotics System	cruise control	KWIC	oscilloscope	Mobile Robotics System
22	Mobile Robotics System provide both deliberative and _____ behavior.	reactive	uncertainty	active	passive	reactive
23	System must be _____ with respect to experimentation and reconfiguration of robot and modification of tasks	feedback	uncertain	discrete	flexible	flexible
24	_____ override currently executing task in sub tree that causes the exception	Exceptions	closedloop feedback	wiretapping	iteration	Exceptions

25	In _____ tasks can eavesdrop on messages intended for other tasks	Exceptions	monitors	wiretapping	reaction to events	wiretapping
26	_____ reads information and execute action if data meets some criterion	Exceptions	monitors	wiretapping	reaction to events	monitors
27	The _____ design is based on hierarchies of tasks or task trees	Exceptions	Wiretapping	Monitors	TCA	TCA
28	In mobile robotics The Level 1 or core layers is used for _____	control routines	schedule	supports concurrency	modeling	control routines
29	In CODGER system blackboard architecture the Pilot component is used for _____	overall supervisor	high-level path planner	low-level path planner	monitors environment	low-level path planner
30	In CODGER system blackboard architecture the Map navigator component is used for _____	overall supervisor	high-level path planner	low-level path planner	monitors environment	high-level path planner
31	In CODGER system blackboard architecture the Lookout component is used for _____	overall supervisor	high-level path planner	low-level path planner	monitors environment	monitors environment
32	A _____system exists to maintain the speed of a car, even over varying terrain	System on/off	cruise control	Engine on/off	Increase/Decrease Speed	cruise control
33	In cruise control Throttle is _____	constant speed	vehicle load	air resistance	Digital value for engine	Digital value for engine
34	In cruise control Pulses is _____	turns the car's wheel	throttle setting	wheel revolution	controlling	wheel revolution
35	In object view of cruise control each blob represents _____	wheel pulses	throttle setting	data	objects	objects
36	For the cruise control, the data element _____ represents the current speed of the vehicle.	Controlled variable	Manipulated variable	Set point	sensors	Controlled variable
37	For the cruise control, the data element _____ represents the throttle setting.	Controlled variable	Manipulated variable	Set point	sensors	Manipulated variable
38	In cruise control, the system is _____ whenever the engine is off	completely off	active	inactive	resume	completely off
39	The PROVOX system by Fisher Controls offers distributed process control for _____ processes	chemical production	leather production	oil production	food production	chemical production

40	The system architecture integrates process control with plant management and information systems in a _____	5-level layered hierarchy	Process measurement	Process supervision	Process management	5-level layered hierarchy
41	_____ is used for direct adjustment of final control elements	5-level layered hierarchy	Process measurement and control	Process supervision	Process management	Process measurement and control
42	_____ is used for operations console for monitoring and controlling	5-level layered hierarchy	Process measurement and control	Process supervision	Process management	Process supervision
43	_____ is used for computer-based plant automation, including management reports, optimization strategies, and guidance to operations console	5-level layered hierarchy	Process measurement and control	Process supervision	Process management	Process management
44	_____ manages higher-level functions such as cost accounting, inventory control, and order processing/scheduling.	Plant and corporate management	Process measurement and control	Process supervision	Process management	Plant and corporate management
45	_____ are current process value, setpoint (target value), valve output, and mode	control points	Operating parameters	Tuning parameters	configuration parameters	Operating parameters
46	_____ are gain, reset, derivative, and alarm trip-points	control points	Operating parameters	Tuning parameters	configuration parameters	Tuning parameters
47	_____ includes tag name and I/O channels	control points	Operating parameters	Tuning parameters	configuration parameters	configuration parameters
48	Rule-based systems provide a means of codifying the _____ skills of human experts.	problem-solving	situation-action rules	analyzing	interpretation	problem-solving
49	_____ surveyed the architecture and operation of rule-based systems.	Hayes-Roth	provox	pranas	mary shaw	Hayes-Roth
50	In Hayes Roth Rule based system the pseudo code is _____	knowledge base	rule interpreter	working memory	rule and data element selector	knowledge base
51	In Hayes Roth Rule based system the interpretation engine is _____	knowledge base	rule interpreter	working memory	rule and data element selector	rule interpreter

52	In Hayes Roth Rule based system the control state of the interpretation engine is _____	knowledge base	rule interpreter	working memory	rule and data element selector	rule and data element selector
53	In Hayes Roth Rule based system the current state of the program is _____	knowledge base	rule interpreter	working memory	rule and data element selector	working memory
54	Rule-based systems make heavy use of _____ and context	a rule executor	Knowledge base	pattern matching	Data Flow	pattern matching
55	In basic rule based system _____ has two components rule base and fact memory	knowledge base	rule interpreter	working memory	rule and data element selector	knowledge base
56	The first major blackboard system was the	HEARSAY-II	HEARSAY-II speech recognition system	KWIC	mobile robotics	HEARSAY-II speech recognition system
57	Each knowledge source is organized as a condition part that specifies _____	blackboard monitor	a scheduler	process	when it is applicable	when it is applicable
58	In Blackboard view of Hearsay -II the _____is realized as a blackboard monitor and a scheduler	Blackboard	knowledge sources	control component	control flow	control component
59	The view as an _____ is a different aggregation of components from the view as blackboard of Hearsey II	program State	pseudo code	control state	interpreter	interpreter
60	The _____monitors the blackboard and calculates priorities for applying the knowledge sources to various elements on the blackboard.	knowledge sources	scheduler	control component	control flow	scheduler

UNIT-II

SYLLABUS

Case studies - Key word is Context – Instrumentation Software – Mobile Robotics – Cruise Control – Three Vignettes in Mixed Style

A KEYWORD IN CONTEXT (KWIC)

This case study shows how different architectural solutions to the same problem provide different benefits.

Parnas proposed the following problems:

KWIC index system accepts an ordered set of lines. Each line is an ordered set of words and each word is an ordered set of characters. Any line may be circularly shifted by repeated removing the first word and appending it at the end of the line. KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

Parnas used the problem to contrast different criteria for decomposing a system into modules.

He describes 2 solutions:
a) Based on functional decomposition with share access to data representation.
b) Based on decomposition that hides design decision.

From the point of view of Software Architecture, the problem is to illustrate the effect of changes on software design. He shows that different problem decomposition vary greatly in their ability to withstand design changes. The changes that are considered by parnas are:

1. The changes in processing algorithm:

Eg: line shifting can be performed on each line as it is read from input device, on all lines after they are read or an demand when alphabetization requires a new set of shifted lines.

2. Changes in data representation:

Eg: Lines, words, characters can be stored in different ways. Circular shifts can be stored explicitly or implicitly Garlan, Kaiser and Notkin also use KWIC problem to illustrate modularization schemes based on implicit invocation. They considered the following.

3. Enhancement to system function:

Modify the system to eliminate circular shift that starts with certain noise change the system to interactive.

4. Performance:

Both space and time

5. Reuse:

Extent to which components serve as reusable entities

Let's outline 4 architectural designs for KWIC system.

SOLUTION 1: MAIN PROGRAM/SUBROUTINE WITH SHARED DATA

- Decompose the problem according to 4 basic functions performed.
  - o Input
  - o Shift
  - o Alphabetize
  - o output
- These computational components are coordinated as subroutines by a main program that sequence through them in turn.
- Data is communicated between components through shared storage.
- Communication between computational component and shared data is constrained by read-write protocol.

Advantages:

- Allows data to be represented efficiently. Since, computation can share the same storage

Disadvantages:

- Change in data storage format will affect almost all of the modules.
- Changes in the overall processing algorithm and enhancement to system function are not easily accommodated.
- This decomposition is not particularly support reuse.

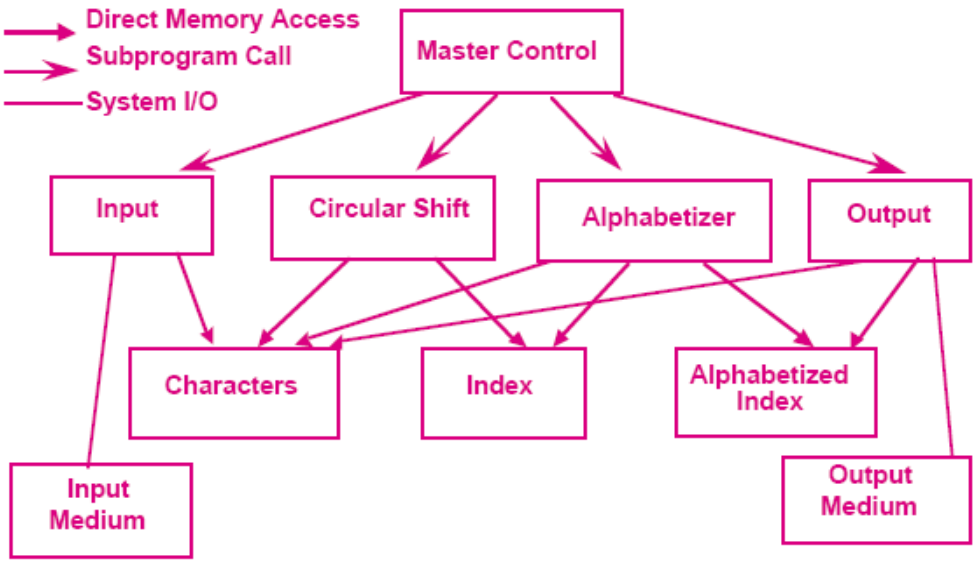


Figure 6: KWIC – Shared Data Solution

SOLUTION 2: ABSTRACT DATA TYPES

- Decomposes The System Into A Similar Set Of Five Modules.
- Data is no longer directly shared by the computational components.
- Each module provides an interface that permits other components to access data only by invoking procedures in that interface.

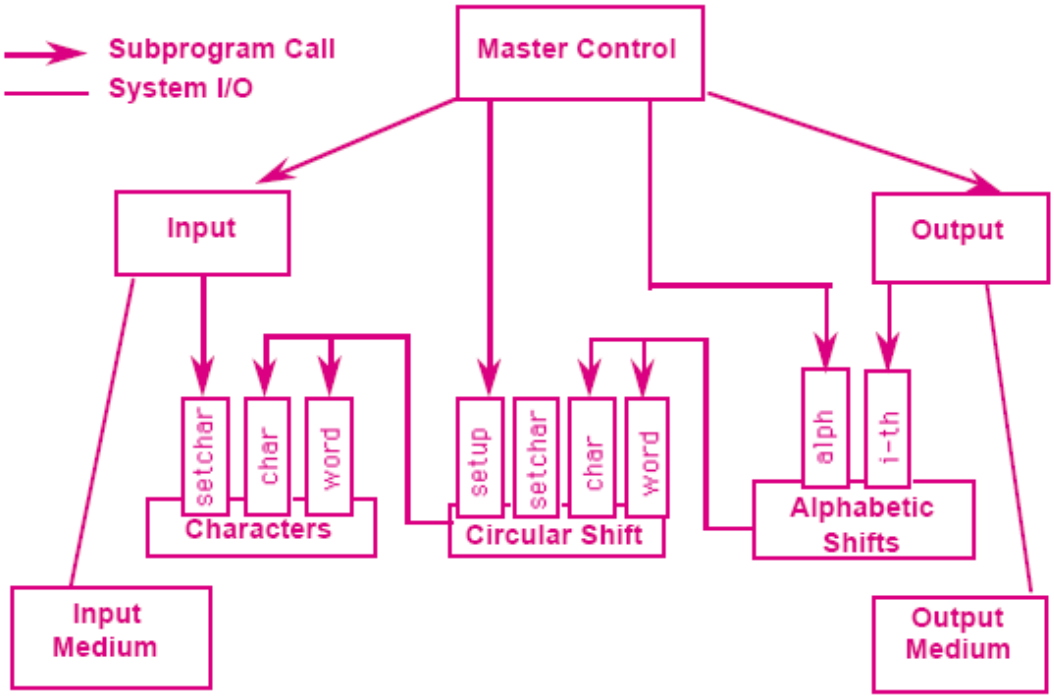


Figure 7: KWIC – Abstract Data Type Solution

Advantage:

- Both Algorithms and data representation can be changed in individual modules without affecting others.
- Reuse is better supported because modules make fewer assumption about the others with which they interact.

Disadvantage:

- Not well suited for functional enhancements
- To add new functions to the system
- To modify the existing modules.

SOLUTION 3: IMPLICIT INVOCATION

- Uses a form of component integration based on shared data
- Differs from 1st solution by these two factors

  o Interface to the data is abstract

o Computations are invoked implicitly as data is modified. Interactions is based on an active data model.

Advantages:

- Supports functional enhancement to the system
- Supports reuse.

Disadvantages:

- Difficult to control the processing order.
- Because invocations are data driven, implementation of this kind of decomposition uses more space.

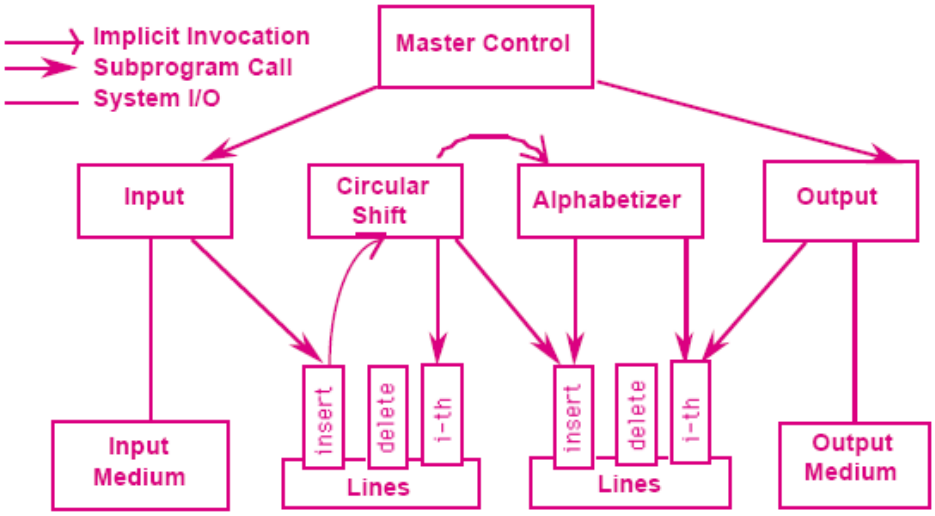


Figure 8: KWIC – Implicit Invocation Solution

SOLUTION 4: PIPES AND FILTERS:

- Four filters: Input, Output, Shift and alphabetize
- Each filter process the data and sends it to the next filter
- Control is distributed

  o Each filter can run whenever it has data on which to compute.

- Data sharing between filters are strictly limited.

Advantages:

- It maintains initiative flow of processing
- It supports reuse
- New functions can be easily added to the system by inserting filters at appropriate level.
- It is easy to modify.

Disadvantages:

- Impossible to modify the design to support an interactive system.
- Solution uses space inefficiently.

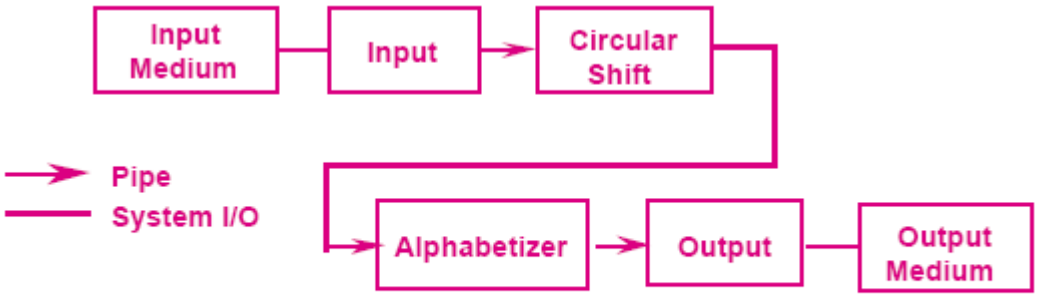


Figure 9: KWIC – Pipe and Filter Solution

	Shared Memory	ADT	Events	Dataflow
Change in Algorithm	–	–	+	+
Change in Data Repn	–	+	–	–
Change in Function	+	–	+	+
Performance	+	+	–	–
Reuse	–	+	–	+

Figure 10: KWIC – Comparison of Solutions

INSTRUMENTATION SOFTWARE:

- Describes the industrial development of software architecture.
- The purpose of the project was to develop a reusable system architecture for oscilloscope
- Oscilloscope is an instrumentation system that samples electrical signals and displays pictures of them on screen.
- Oscilloscope also performs measurements on the signals and displays them on screen.
- Modern oscilloscope has to perform dozens of measurements supply megabytes of internal storage.
- Support an interface to a network of workstations and other instruments and provide sophisticated
- user interface, including touch panel screen with menus, built-in help facilities and color displays.
- Problems faced:
- Little reuse across different oscilloscope products.
- Performance problems were increasing because the software was not rapidly configurable
- within the instrument.
- Goal of the project was to develop an architectural framework for oscilloscope.
- Result of that was domain specific software architecture that formed the basis of the next generation of oscilloscopes.

SOLUTION 1: OBJECT ORIENTED MODEL

Different data types used in oscilloscope are:

- Waveforms
- Signals
- Measurements
- Trigger modes so on

There was no overall model that explained how the types fit together. This led to confusion about the partitioning of functionality.

Ex: it is not clearly defined that measurements to be associated with types of data being measured or represented externally.

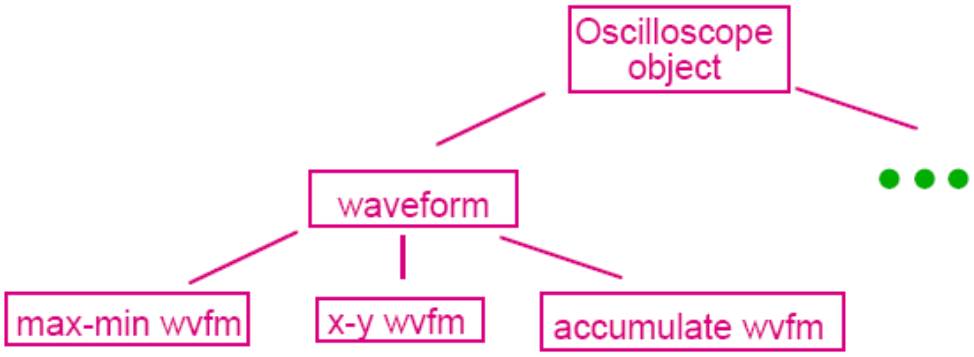


Figure 11: Oscilloscopes – An Object-oriented Model

SOLUTION 2: LAYERED MODEL

- To correct the problems by providing a layered model of an oscilloscope.
- Core-layer: implemented in hardware represents signal manipulation functions that filter signals as they enter the oscilloscope.
- Initially the layered model was appealing since it partitioned the functions of an oscilloscope into well defined groups.
- But, it was a wrong model for the application domain. Because, the problem was that the boundaries of abstraction enforced by the layers conflicted with the needs for interaction among various functions.

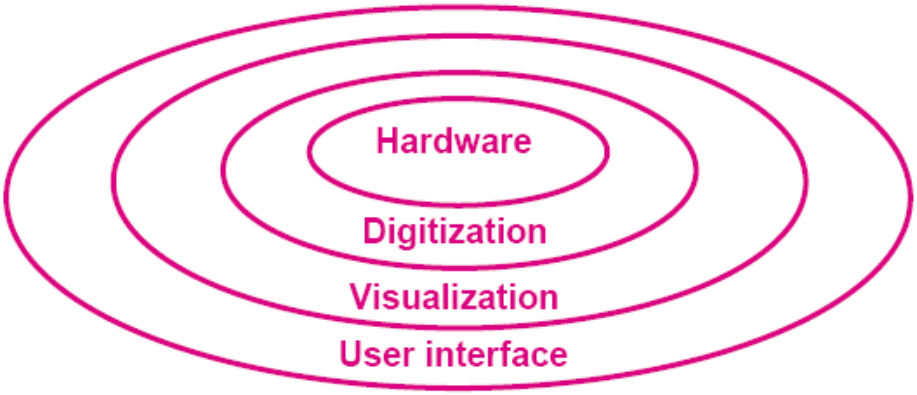


Figure 12: Oscilloscopes – A Layered Model

SOLUTION 3: PIPE-AND-FILTER MODEL:

- In this approach oscilloscope functions were viewed as incremental transformers of data.
  - o Signal transformer: to condition external signal.
  - o Acquisition transformer: to derive digitized waveforms
  - o Display transformers: to convert waveforms into visual data.
- It is improvement over layered model as it did not isolate the functions in separate partition.
- Main problem with this model is that
  - o It is not clear how the user should interact with it.

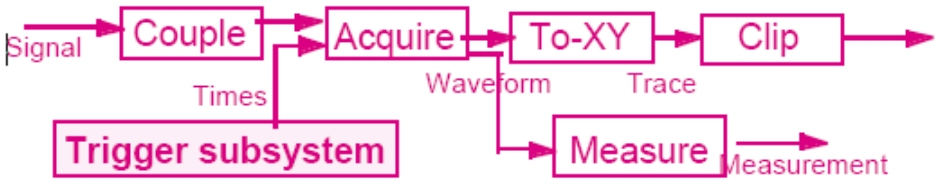


Figure 13: Oscilloscopes – A Pipe and Filter Model

SOLUTION 4: MODIFIED PIPE-AND-FILTER MODEL:

To overcome the above said problem, associate control interface with each filter that allowed external entity to set parameters of operation for the filter.

Introduction of control interface solves a large part of the user interface problem

- It provides collection of setting that determines what aspect of the oscilloscope can be modified
- dynamically by the user.
- It explains how user can change functions by incremental adjustments to the software.

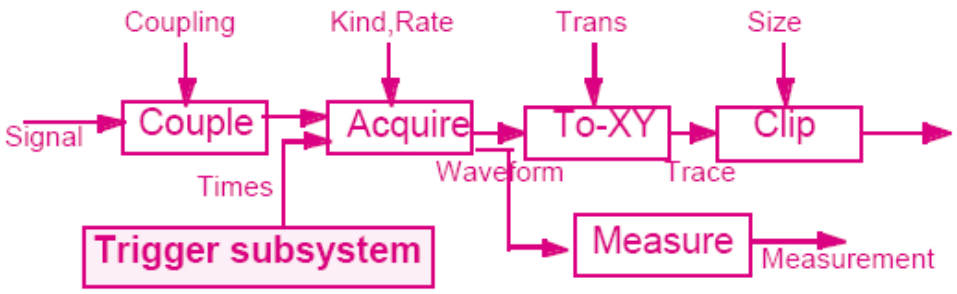


Figure 14: Oscilloscopes – A Modified Pipe and Filter Model

FURTHER SPECIALIZATION

The above described model is greater improvement over the past. But, the main problem with this is the performance.

a. Because waveform occupy large amount of internal storage

It is not practical for each filter to copy waveforms every time they process them.

b. Different filters run at different speeds

It is unacceptable to slow one filter down because another filter is still processing its data.

To overcome the above discussed problems the model is further specialized.

Instead of using same kind of pipe. We use different "colors" of pipe. To allow data to be processed without copying, slow filters to ignore incoming data.

These additional pipes increased the stylistic vocabulary and allowed pipe/filter computations to be tailored more specifically to the performance needs of the product.

MOBILE ROBOTICS

Mobile Robotic systems

- Controls a manned or semi-manned vehicle

o E.g., car, space vehicle, etc

- Used in space exploration missions
- Hazardous waste disposal
- Underwater exploration

The system is complex

- Real Time respond
- input from various sensors
- Controlling the motion and movement of robots
- Planning its future path/move

Unpredictability of environment

- Obstacles blocking robot path
- Sensor may be imperfect
- Power consumption

Respond to hazardous material and situations

DESIGN CONSIDERATIONS

REQ1: Supports deliberate and reactive behavior. Robot must coordinate the actions to accomplish its

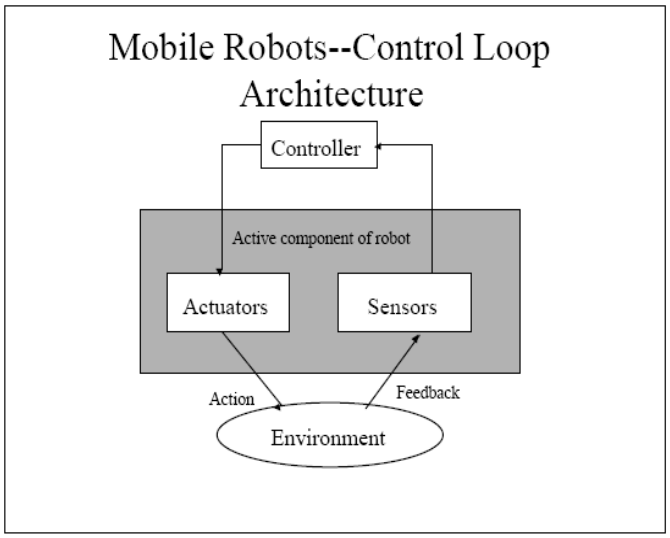
mission and reactions to unexpected situations

REQ2: Allows uncertainty and unpredictability of environment. The situations are not fully defined

and/or predicable. The design should handle incomplete and unreliable information

REQ3: System must consider possible dangerous operations by Robot and environment

REQ4: The system must give the designer flexibility (mission's change/requirement changes)



SOLUTION 1: CONTROL LOOP

□□**Req1:** an advantage of the closed loop paradigm is its simplicity □□it captures the basic interaction between the robot and the outside.

□□**Req2:** control loop paradigm is biased towards one method □□reducing the unknowns through iteration

□□**Req3:** fault tolerance and safety are supported which makes duplication easy and reduces the chances of errors

□□**Req4:** the major components of a robot architecture are separated from each other and can be replaced independently

SOLUTION 2: LAYERED ARCHITECTURE

Figure shows Alberto Elfes's definition of the layered architecture.

□□Level 1 (core) control routines (motors, joints,..),

□□Level 2-3 real world I/P (sensor interpretation and integration

(analysis of combined I/Ps)

□□Level 4 maintains the real world model for robot

□□Level 5 manage navigation

□□Level 6-7 Schedule & plan robot actions (including exception

handling and re-planning)

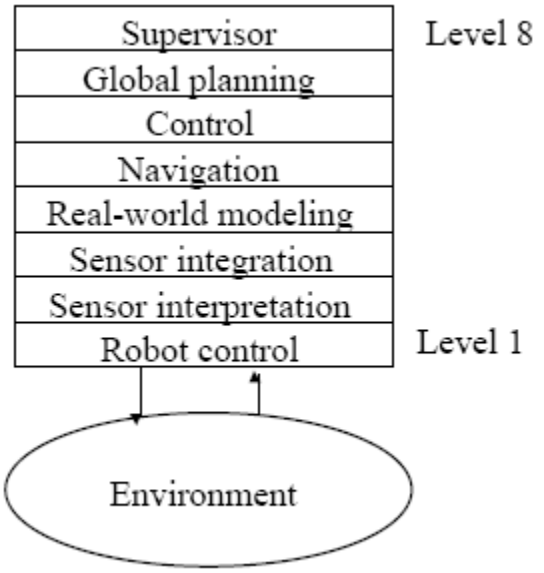
Top level deals with UI and overall supervisory functions

Req1: it overcomes the limitations of control loop and it defines abstraction levels to guide the design

Req2: uncertainty is managed by abstraction layers

Req3: fault tolerance and passive safety are also served

Req4: the interlayer dependencies are an obstacle to easy replacement and addition of components.



SOLUTION 3: IMPLICIT INVOCATION

The third solution is based on the form of implicit invocation, as embodied in the Task-Control-Architecture (TCA). The TCA design is based on hierarchies of tasks or task trees

Parent tasks initiate child task

Temporal dependencies between pairs of tasks can be defined

A must complete A must complete before B starts (selective concurrency)

Allows dynamic reconfiguration of task tree at run time in response to sudden change(robot and environment)

Uses implicit invocation to coordinate tasks

 Tasks communicate using multicasting message (message server) to tasks that are registered for these events TCA's implicit invocation mechanisms support three functions:

Exceptions: Certain conditions cause the execution of an associated exception handling routines

 i.e., exception override the currently executing task in the sub-tree (e.g., abort or retry) tasks

Wiretapping: Message can be intercepted by tasks superimposed on an existing task tree

 E.g., a safety-check component utilizes this to validate outgoing motion commands

Monitors: Monitors read information and execute some action if the data satisfy certain condition

 E.g. battery check

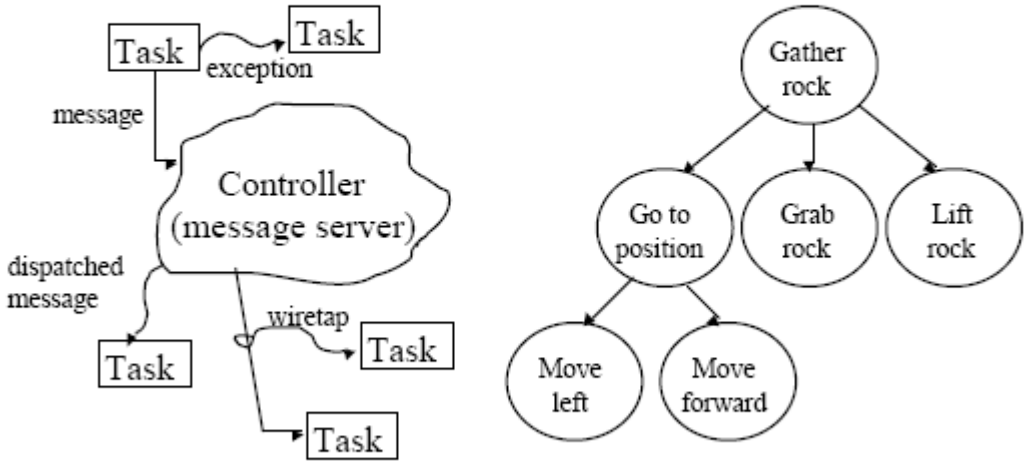
Req1: permits clear cut separation of action and reaction

Req2: a tentative task tree can be built to handle uncertainty

Req3: performance, safety and fault tolerance are served

Req4: makes incremental development and replacement of components straight forward

Mobile Robots--Task-control Architecture (Implicit Invocation)



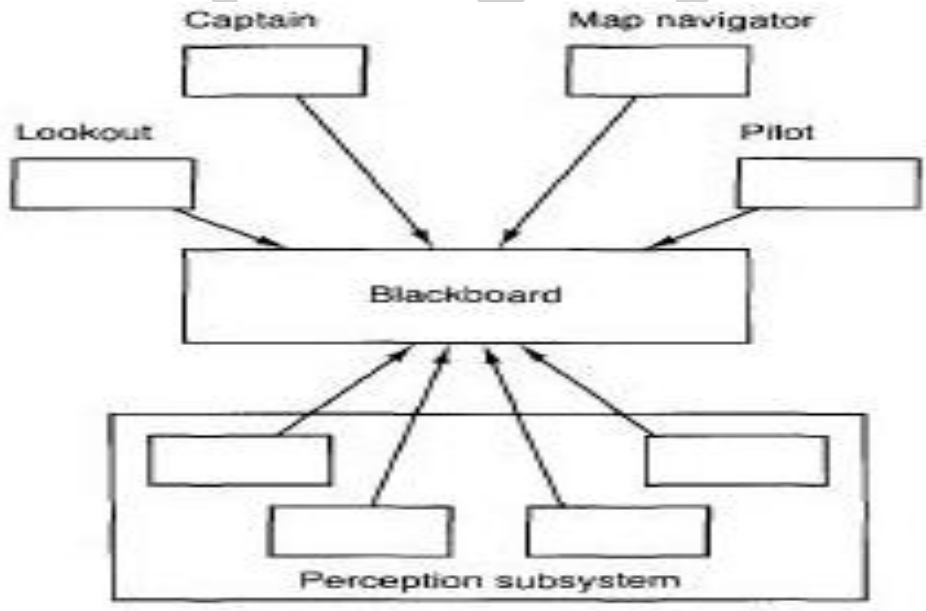
SOLUTION 4: BLACKBOARD ARCHITECTURE

The components of CODGER are the following:

- Captain: overall supervisor
- Map navigator: high-level path planner
- Lookout: monitors environment for landmarks
- Pilot: low-level path planner and motor controller
- Perception subsystems: accept sensor input and integrate it into a coherent situation interpretation

The requirements are as follows:

- **Req1:** the components communicate via shared repository of the blackboard system.
- **Req2:** the blackboard is also the means for resolving conflicts or uncertainties in the robot's world view
- **Req3:** speed, safety and reliability is guaranteed
- **Req4:** supports concurrency and decouples senders from receivers, thus facilitating maintenance. **Figure: blackboard solution for mobile robots**



COMPARISONS

	Control Loop	Layers	Impl. Invoc.	Black Board
Task Coordination	+–	–	++	+
Dealing with Uncertainty	–	+–	+–	+
Fault Tolerance	+–	+–	++	+
Safety	+–	+–	++	+
Performance	+–	+–	++	+
Flexibility	+–	–	+	+

Table 2.2.1. Strengths and Weaknesses of Robot Architectures

CRUISE CONTROL

A cruise control (CC) system that exists to maintain the constant vehicle speed even over varying terrain.

Inputs:

System On/Off: If on, maintain speed

Engine On/Off: If on, engine is on. CC is active only in this state

Wheel Pulses: One pulse from every wheel revolution

Accelerator: Indication of how far accelerator is de-pressed

Brake: If on, temp revert cruise control to manual mode

Inc/Dec Speed: If on, increase/decrease maintained speed

Resume Speed: If on, resume last maintained speed

Clock: Timing pulses every millisecond

Outputs:

Throttle: Digital value for engine throttle setting

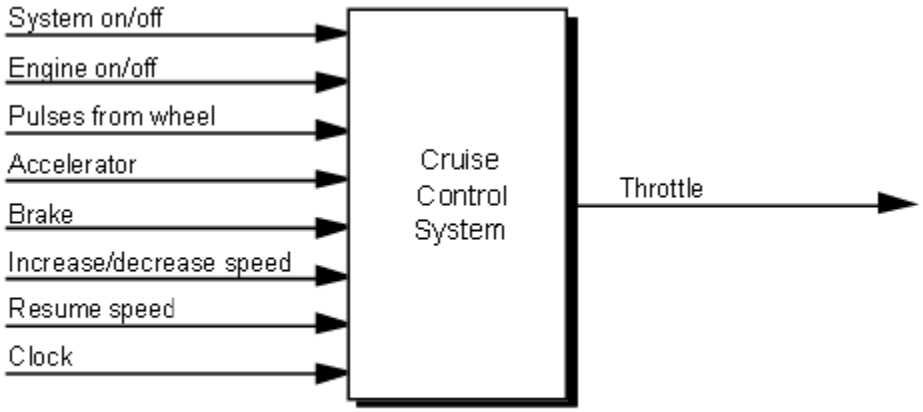


Figure 5: Booch block diagram for cruise control

Restatement of Cruise-Control Problem

Whenever the system is active, determine the desired speed, and control the engine throttle setting to maintain that speed.

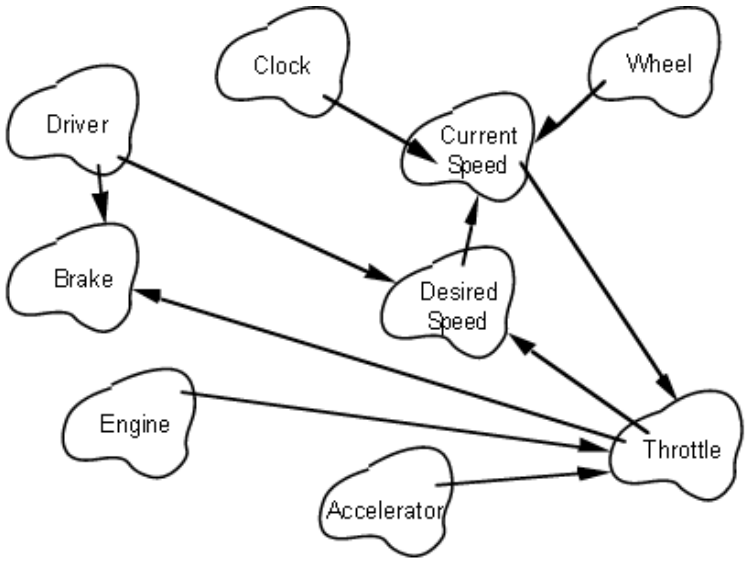
OBJECT VIEW OF CRUISE CONTROL

□□Each element corresponds to important quantities and physical entities in the system

□□Each blob represents objects

□□Each directed line represents dependencies among the objects

The figure corresponds to Booch's object oriented design for cruise control



PROCESS CONTROL VIEW OF CRUISE CONTROL

□□□**Computational Elements**

□□*Process definition* - take throttle setting as I/P & control vehicle speed

□□*Control algorithm* - current speed (wheel pulses) compared to desired speed

o Change throttle setting accordingly presents the issue:

o decide how much to change setting for a given discrepancy

□□**Data Elements**

□□*Controlled variable:* current speed of vehicle

□□*Manipulated variable:* throttle setting

□□*Set point:* set by accelerator and increase/decrease speed inputs

□□system on/off, engine on/off, brake and resume inputs also have a bearing

□□*Controlled variable sensor*: modelled on data from wheel pulses and clock

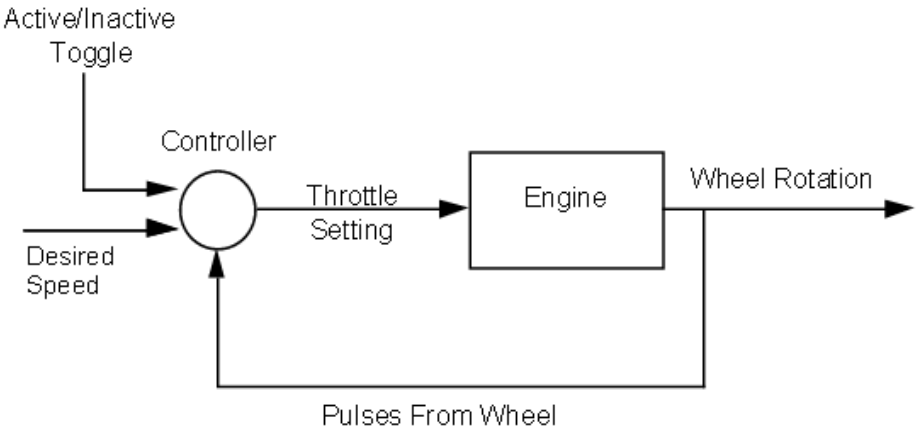


Figure 7: Control Architecture for Cruise Control

Figure 3.18 control architecture for cruise control

The active/inactive toggle is triggered by a variety of events, so a state transition design is natural. It's shown in Figure. The system is completely off whenever the engine is off. Otherwise there are three inactive and one active state.

For simplicity we assume brake application is atomic so other events are blocked when the brake is on. A more detailed analysis of the system states would relax this Assumption.

The active/inactive toggle is triggered by a variety of events, so a state transition design is natural. It's shown in Figure 8. The system is completely off whenever the engine is off. Otherwise there are three inactive and one active states. In the first inactive state no set point has been established. In the other two, the previous set point must be remembered:

When the driver accelerates to a speed greater than the set point, the manual Accelerator controls the throttle through a direct linkage (note that this is the only use of the accelerator position in this design, and it relies on relative effect rather than absolute position); when the driver uses the brake the control system is inactivated until the resume signal is sent. The active/inactive toggle input of the control system is set to active exactly when this state machine is in state Active

Determining the desired speed is simpler, since it does not require state other than the current value of desired speed (the set point). Any time the system is off, the set point is undefined. Any time the system on signal is given (including when the system is already on) the set point is set to the current speed as modeled by wheel pulses.

The driver also has a control that increases or decreases the set point by a set amount. This, too, can be invoked at any time (define arithmetic on undefined values to yield undefined values). Figure 9 summarizes the events involved in determining the set point. Note that this process requires access to the clock in order to estimate the current speed based on the pulses from the wheel.

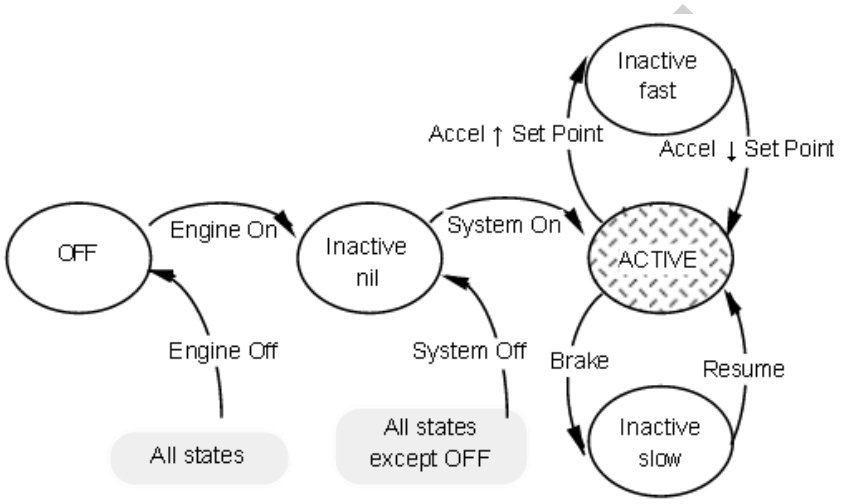


Figure 8: State Machine for Activation

Figure state machine for activation

Event	*Effect on desired speed*
Engine off, system off	Set to "undefined"
System on	Set to current speed as estimated from wheel pulses
Increase speed	Increment desired speed by constant
Decrease speed	Decrement desired speed by constant

Figure 9: Event Table for Determining Set Point

We can now combine the control architecture, the state machine for activation, and the event table for determining the set point into an entire system.

We can now compose the control architecture, the state machine for activation, and the event table for determining the set point into an entire system. Although there is no need for the control unit and set point determination to use the same clock, we do so to minimize changes to the original problem statement. Then,

since current speed is used in two components, it would be reasonable for the next elaboration of the design to encapsulate that model in a reusable object; this would encapsulate the clock. All of the objects in Booch's design (Figure 6) have clear roles in the resulting system. It is entirely reasonable to look forward to a design strategy in which the control loop architecture is used for the system as a whole and a number of other architectures, including objects and state machines, are used in the elaborations of the elements of the control loop architecture.

The shift from an object-oriented view to a control view of the cruise control architecture raised a number of design questions that had previously been slighted: The separation of process from control concerns led to explicit choice of the control discipline. The limitations of the control model also became clear, including possible inaccuracies in the current speed model and incomplete control at high speed. The dataflow character of the model showed irregularities in the way the input was specified, for example mixture of state and event inputs and the inappropriateness of absolute position of the accelerator

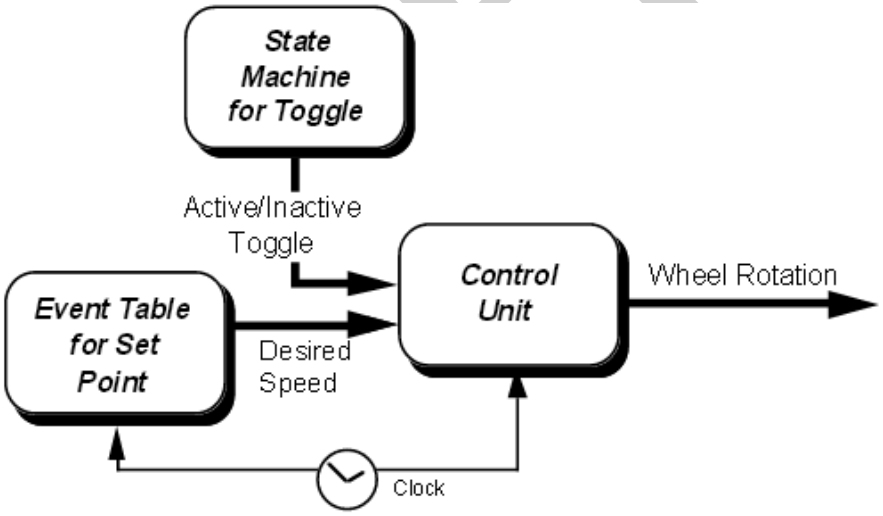


Figure 10: Complete cruise control system

Figure 3.21 complete cruise control system

THREE VIGNETTES IN MIXED STYLE

A LAYERED DESIGN WITH DIFFERENT STYLES FOR THE LAYERS

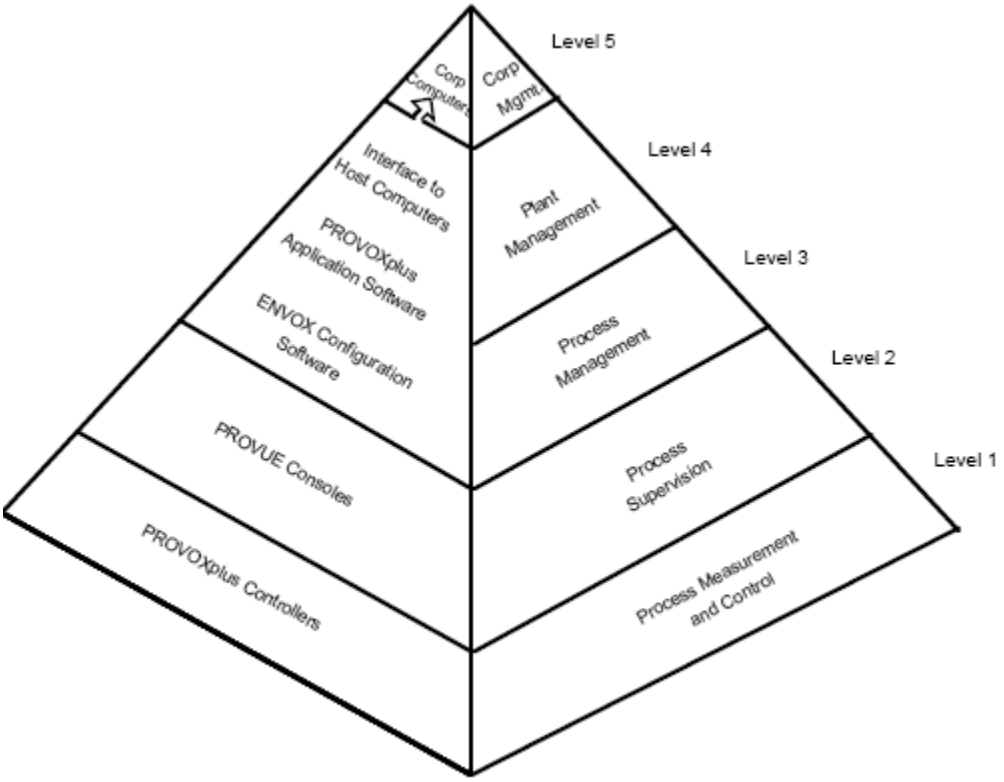


Figure 19: PROVOX – Hierarchical Top Level

Each level corresponds to a different process management function with its own decision-support requirements.

Level 1: Process measurement and control: direct adjustment of final control elements.

Level 2: Process supervision: operations console for monitoring and controlling Level 1.

Level 3: Process management: computer-based plant automation, including management reports, optimization strategies, and guidance to operations console.

Levels 4 and 5: Plant and corporate management: higher-level functions such as cost accounting, inventory control, and order processing/scheduling.

Figure 20 shows the canonical form of a point definition; seven specialized forms support the most common kinds of control. Points are, in essence, object-oriented design elements that encapsulate information about control points of the process. Data associated with a point includes: Operating parameters, including current process value, set point (target value), valve output, and mode (automatic or manual); Tuning parameters, such as gain, reset, derivative, and alarm trip-points; Configuration parameters, including tag (name) and I/O channels.

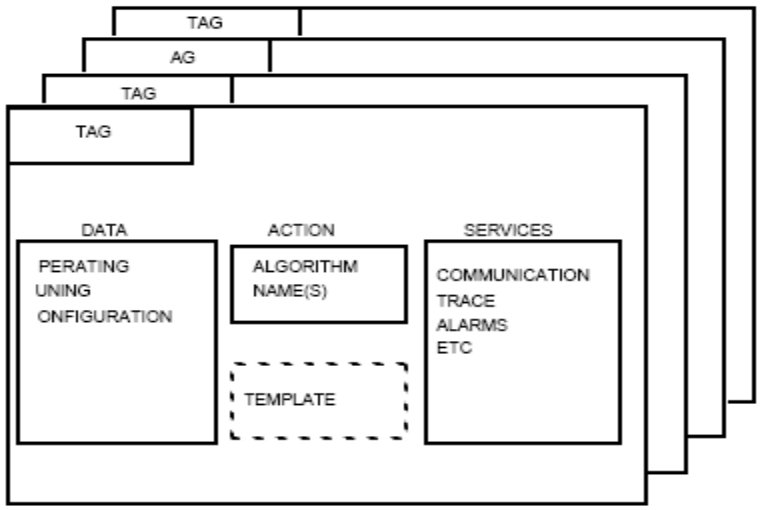


Figure 20: PROVOX – Object-oriented Elaboration

AN INTERPRETER USING DIFFERENT IDIOMS FOR THE COMPONENTS

Rule-based systems provide a means of codifying the problem-solving knowhow of human experts. These experts tend to capture problem-solving techniques as sets of situation-action rules whose execution or activation is sequenced in response to the conditions of the computation rather than by a predetermined scheme. Since these rules are not directly executable by available computers, systems for interpreting such rules must be provided. Hayes-Roth surveyed the architecture and operation of rule-based systems.

The basic features of a rule-based system, shown in Hayes-Roth's rendering as Figure 21, are essentially the features of a table-driven interpreter, as outlined earlier.

• The *pseudo-code* to be executed, in this case the knowledge base

• The *interpretation engine,* in this case the rule interpreter, the heart of the inference engine

• The *control state of the interpretation engine,* in this case the rule and data element selector

• The *current state of the program* running on the virtual machine, in this case the working memory.

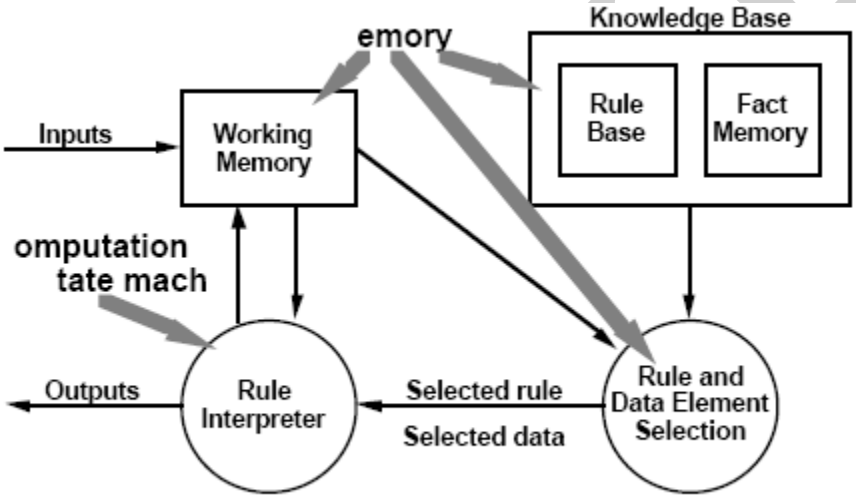


Figure 21: Basic Rule-Based System

Rule-based systems make heavy use of pattern matching and context (currently relevant rules). Adding special mechanisms for these facilities to the design leads to the more complicated view shown in Figure 22.

We see that:

• The knowledge base remains a relatively simple memory structure, merely gaining substructure to distinguish active from inactive contents.

• The rule interpreter is expanded with the interpreter idiom, with control procedures playing the role of the pseudo-code to be executed and the execution stack the role of the current program state.

• "Rule and data element selection" is implemented primarily as a pipeline that progressively transforms active rules and facts to prioritized activations.

• Working memory is not further elaborated.

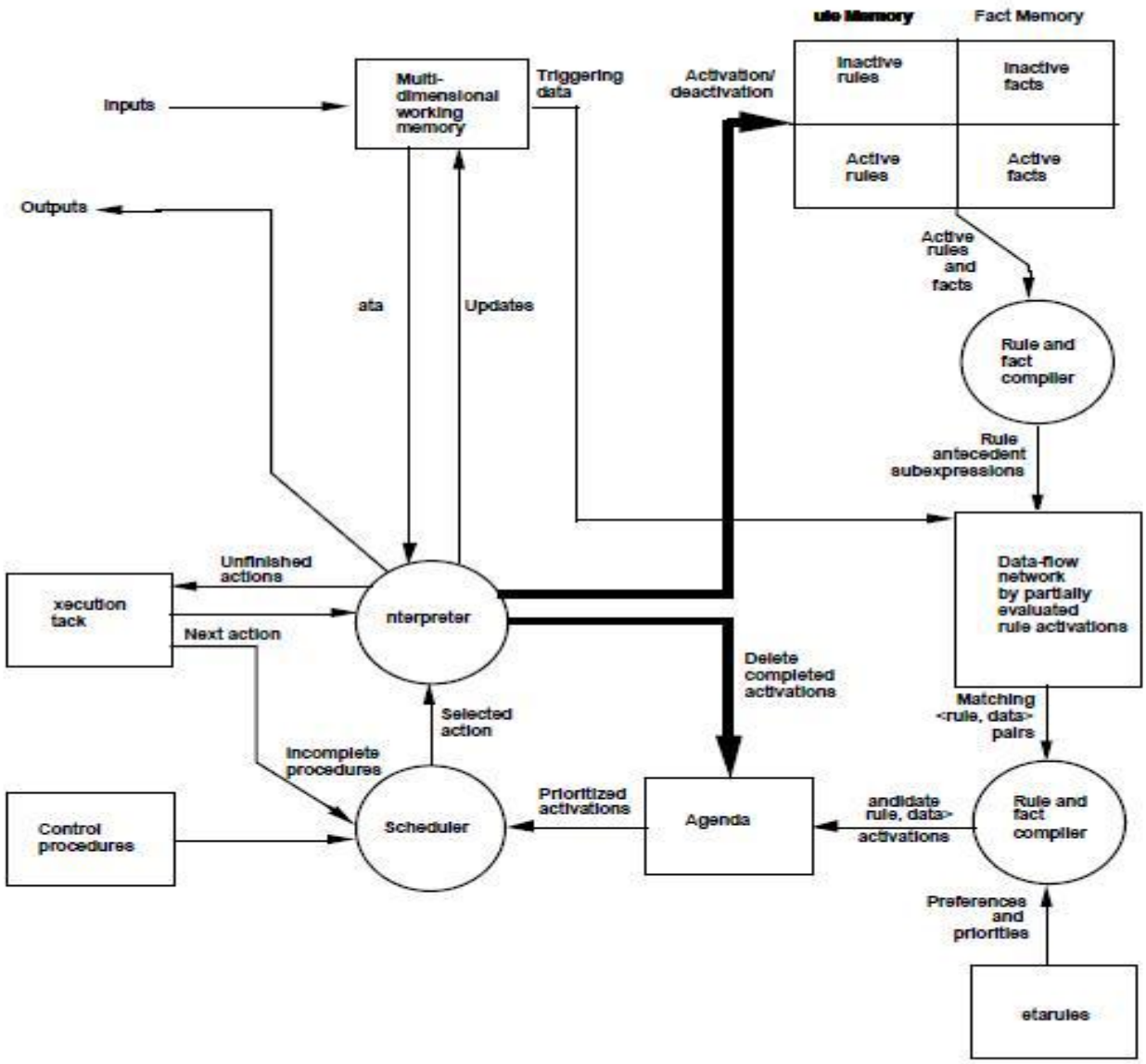


Figure 23: Sophisticated Rule-Based System

A BLACKBOARD GLOBALLY RECAST AS AN INTERPRETER

The blackboard model of problem solving is a highly structured special case of opportunistic problem solving. In this model, the solution space is organized into several application dependent hierarchies and the domain knowledge is partitioned into independent modules of knowledge that operate on knowledge within and between levels.

Figure showed the basic architecture of a blackboard system and outlined its three major parts: knowledge sources, the blackboard data structure, and control.

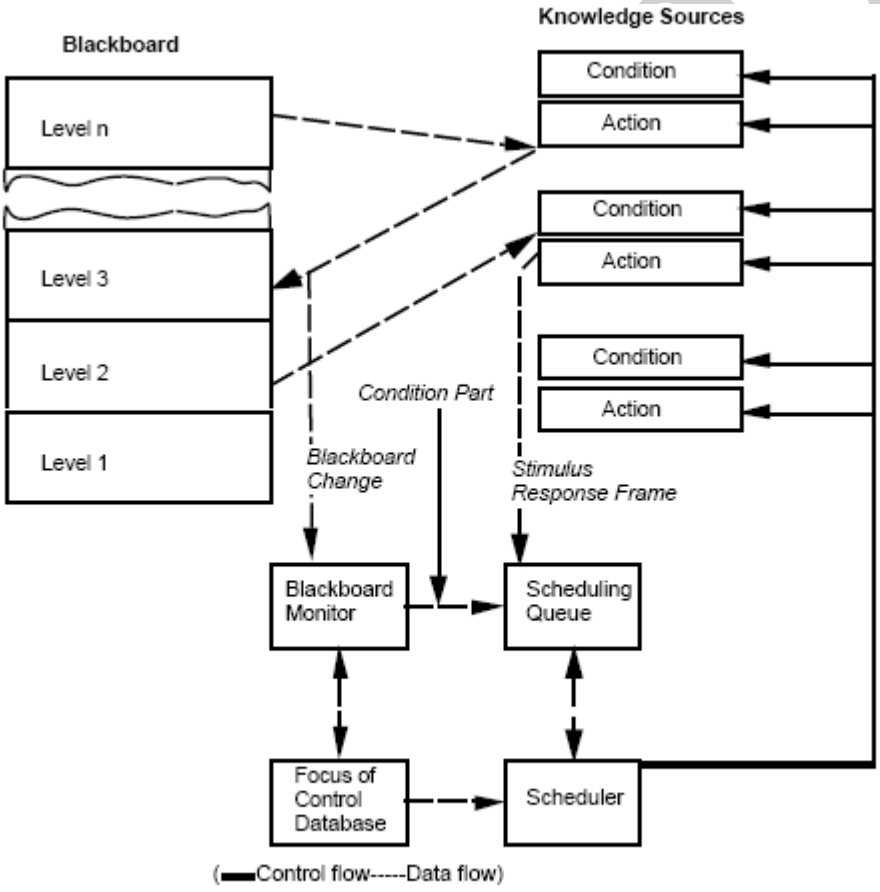


Figure 24: Hearsay-II

The first major blackboard system was the HEARSAY-II speech recognition system. Nii's schematic of the HEARSAY-II architecture appears as Figure 24. The blackboard structure is a six- to eight-level hierarchy in which each level abstracts information on its adjacent lower level

and blackboard elements represent hypotheses about the interpretation of an utterance. HEARSAY-II was implemented between 1971 and 1976; these machines were not directly capable of condition-triggered control, so it should not be surprising to find that an implementation provides the mechanisms of a virtual machine that realizes the implicit invocation semantics required by the blackboard model.

POSSIBLE QUESTIONS

PART – B

1. Discuss about Key Word in Context with a neat diagram.
2. Discuss about the Instrumentation Software with neat diagram.
3. Explain the Mobile Robotics System in detail.
4. What are the key characteristics of Cruise Control? Explain.
5. Elaborate on Oscilloscope i) Pipe and Filter ii) A Modified Pipe and Filter Model with example.
6. Discuss about mobile Three Vignettes in Mixed Style with neat diagram..

PART – C

1. Discuss about Key Word in Context with a neat diagram
2. Explain the Mobile Robotics System in detail.
3. Explain Cruise Control in detail.

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS: I MSC CS SUBJECT NAME: INTRODUCTION TO SOFTWARE ARCHITECTURE
SUBJECT CODE: 17CSP204 BATCH-2017-2019

UNIT: II (ONE MARKS) PART A - ONLINE EXAMINATION

S.NO	QUESTION	CHOICE 1	CHOICE 2	CHOICE 3	CHOICE 4	ANSWER
1	The KWIC means	Key Word in Context	Key word in Constant	Key word as Context	Key word as Capital	Key Word in Context
2	The KWIC index system outputs a listing of all _____ of all lines in alphabetical order	circular shifts	ordered set of words	ordered set of characters	set of lines	circular shifts
3	The _____ solution decomposes the problem according to the four basic functions performed: input, shift, alphabetize, and output.	Main program/Subroutine with shared data	abstract data type	layered	event based	Main program/Subroutine with shared data
4	Data is communicated between the components through _____.	shared storage	module	algorithm	data	shared storage
5	Communication between the computational components and the shared data is an unconstrained _____ protocol	read write	network	Communication	read	read write
6	In Abstract Data Types each module provides an interface that permits other components to access data only by invoking _____	shared storage	module	procedures in that interface	data	procedures in that interface
7	In _____ solution computations are invoked implicitly as data is modified.	Implicit Invocation	abstract data type	layered	process control	Implicit Invocation
8	The KWIC index system outputs a listing of all circular shifts of all lines in _____ order.	forward	alphabetical	backward	circle	alphabetical
9	In pipe and filter new functions are easily added to the system by _____ at the appropriate point in the processing sequence	inserting filter	editing filter	deleting filter	copying filter	inserting filter
10	The permuted" [sic] index for the Unix Man pages is an example of _____	KWIC	implicit invocation	instrumentation software	robotics	KWIC
11	The data types used in oscilloscopes is _____	waveforms, signals	int	float	char	waveforms, signals

12	_____ company work towards the software to support their Instrumentation products like oscilloscope	Dell laboratories	Computer Research Laboratory	Tektronix	Cipla	Tektronix
13	An oscilloscope is an instrumentation system that samples electrical signals and displays pictures_____ of them on a screen	traces	pulses	waves	lines	traces
14	In layered model the _____ represented the signal manipulation functions that filter signals as they enter the oscilloscope.	individual modules	core layer	outer layer	inner layer	core layer
15	In _____ layer signals are digitized and stored internally for later processing	waveform acquisition	hardware	user interface	visual	waveform acquisition
16	The outermost layer in oscilloscope's layered model is _____	user interface	waveform acquisition	hardware	visual	user interface
17	_____serve to condition external signals in pipe filter model of oscilloscope	Signal transformers	acquisition transformers	Display transformers	waveform transformers	Signal transformers
18	_____ derive digitized waveforms from these signals in pipe filter model of oscilloscope	Signal transformers	acquisition transformers	Display transformers	waveform transformers	acquisition transformers
19	_____convert these waveforms into visual data in pipe filter model of oscilloscope	Signal transformers	acquisition transformers	Display transformers	waveform transformers	Display transformers
20	The _____ solution accounted for user inputs by associating with each filter a control interface that allows an external entity to set parameters of operation for the filter	A Modified Pipe and Filter Model	robotics	implicit invocation	instrumentation software	A Modified Pipe and Filter Model
21	In _____ controls manned, partially-manned, or unmanned vehicle	Mobile Robotics System	cruise control	KWIC	oscilloscope	Mobile Robotics System
22	Mobile Robotics System provide both deliberative and _____ behavior.	reactive	uncertainty	active	passive	reactive
23	System must be _____ with respect to experimentation and reconfiguration of robot and modification of tasks	feedback	uncertain	discrete	flexible	flexible
24	_____ override currently executing task in sub tree that causes the exception	Exceptions	closedloop feedback	wiretapping	iteration	Exceptions

25	In _____ tasks can eavesdrop on messages intended for other tasks	Exceptions	monitors	wiretapping	reaction to events	wiretapping
26	_____ reads information and execute action if data meets some criterion	Exceptions	monitors	wiretapping	reaction to events	monitors
27	The _____ design is based on hierarchies of tasks or task trees	Exceptions	Wiretapping	Monitors	TCA	TCA
28	In mobile robotics The Level 1 or core layers is used for _____	control routines	schedule	supports concurrency	modeling	control routines
29	In CODGER system blackboard architecture the Pilot component is used for _____	overall supervisor	high-level path planner	low-level path planner	monitors environment	low-level path planner
30	In CODGER system blackboard architecture the Map navigator component is used for _____	overall supervisor	high-level path planner	low-level path planner	monitors environment	high-level path planner
31	In CODGER system blackboard architecture the Lookout component is used for _____	overall supervisor	high-level path planner	low-level path planner	monitors environment	monitors environment
32	A _____system exists to maintain the speed of a car, even over varying terrain	System on/off	cruise control	Engine on/off	Increase/Decrease Speed	cruise control
33	In cruise control Throttle is _____	constant speed	vehicle load	air resistance	Digital value for engine	Digital value for engine
34	In cruise control Pulses is _____	turns the car's wheel	throttle setting	wheel revolution	controlling	wheel revolution
35	In object view of cruise control each blob represents _____	wheel pulses	throttle setting	data	objects	objects
36	For the cruise control, the data element _____ represents the current speed of the vehicle.	Controlled variable	Manipulated variable	Set point	sensors	Controlled variable
37	For the cruise control, the data element _____ represents the throttle setting.	Controlled variable	Manipulated variable	Set point	sensors	Manipulated variable
38	In cruise control, the system is _____ whenever the engine is off	completely off	active	inactive	resume	completely off
39	The PROVOX system by Fisher Controls offers distributed process control for _____ processes	chemical production	leather production	oil production	food production	chemical production

40	The system architecture integrates process control with plant management and information systems in a _____	5-level layered hierarchy	Process measurement	Process supervision	Process management	5-level layered hierarchy
41	_____ is used for direct adjustment of final control elements	5-level layered hierarchy	Process measurement and control	Process supervision	Process management	Process measurement and control
42	_____ is used for operations console for monitoring and controlling	5-level layered hierarchy	Process measurement and control	Process supervision	Process management	Process supervision
43	_____ is used for computer-based plant automation, including management reports, optimization strategies, and guidance to operations console	5-level layered hierarchy	Process measurement and control	Process supervision	Process management	Process management
44	_____ manages higher-level functions such as cost accounting, inventory control, and order processing/scheduling.	Plant and corporate management	Process measurement and control	Process supervision	Process management	Plant and corporate management
45	_____ are current process value, setpoint (target value), valve output, and mode	control points	Operating parameters	Tuning parameters	configuration parameters	Operating parameters
46	_____ are gain, reset, derivative, and alarm trip-points	control points	Operating parameters	Tuning parameters	configuration parameters	Tuning parameters
47	_____ includes tag name and I/O channels	control points	Operating parameters	Tuning parameters	configuration parameters	configuration parameters
48	Rule-based systems provide a means of codifying the _____ skills of human experts.	problem-solving	situation-action rules	analyzing	interpretation	problem-solving
49	_____ surveyed the architecture and operation of rule-based systems.	Hayes-Roth	provox	pranas	mary shaw	Hayes-Roth
50	In Hayes Roth Rule based system the pseudo code is _____	knowledge base	rule interpreter	working memory	rule and data element selector	knowledge base
51	In Hayes Roth Rule based system the interpretation engine is _____	knowledge base	rule interpreter	working memory	rule and data element selector	rule interpreter

52	In Hayes Roth Rule based system the control state of the interpretation engine is _____	knowledge base	rule interpreter	working memory	rule and data element selector	rule and data element selector
53	In Hayes Roth Rule based system the current state of the program is _____	knowledge base	rule interpreter	working memory	rule and data element selector	working memory
54	Rule-based systems make heavy use of _____ and context	a rule executor	Knowledge base	pattern matching	Data Flow	pattern matching
55	In basic rule based system _____ has two components rule base and fact memory	knowledge base	rule interpreter	working memory	rule and data element selector	knowledge base
56	The first major blackboard system was the	HEARSAY-II	HEARSAY-II speech recognition system	KWIC	mobile robotics	HEARSAY-II speech recognition system
57	Each knowledge source is organized as a condition part that specifies _____	blackboard monitor	a scheduler	process	when it is applicable	when it is applicable
58	In Blackboard view of Hearsay -II the _____is realized as a blackboard monitor and a scheduler	Blackboard	knowledge sources	control component	control flow	control component
59	The view as an _____ is a different aggregation of components from the view as blackboard of Hearsey II	program State	pseudo code	control state	interpreter	interpreter
60	The _____monitors the blackboard and calculates priorities for applying the knowledge sources to various elements on the blackboard.	knowledge sources	scheduler	control component	control flow	scheduler

UNIT-III

Shared Information Systems – Database Integration – Integration in Software Development Environments – Integration in the Design of Buildings – Architectural structures for shared Information Systems

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, mainly

 Data Processing: Driven primarily by the need to build business decision systems from conventional databases.

 Software Development Environment: 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.

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
 not tolerant for structural modification
 not responsive to the needs of interactive processing

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.

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. Interactive demands required individual transactions to complete in real time.

Information began to appear redundantly in multiple databases, and geographic distribution added communication complexity.

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.

As databases became more common, information about a business became distributed among multiple databases. Hence data become inconsistent and incomplete. 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
- multi-databases.

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. Atypical organization included:

- a massive *edit program*: accepts transaction inputs and perform validation without accessing the database.
- a massive *transaction sort*: get 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*: produce periodic reports

Batch sequential architecture:

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.

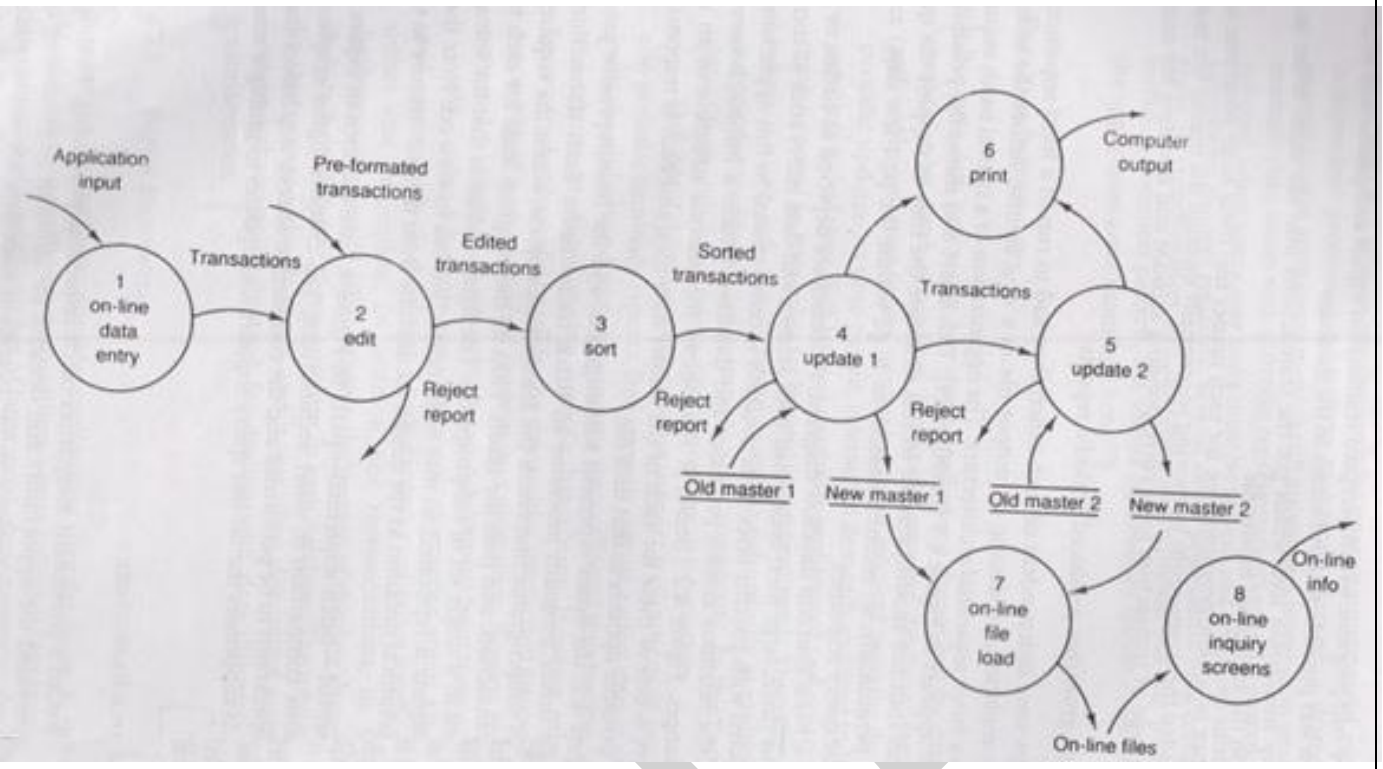


Fig: Data flow diagram for batch databases

The above figure 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.

Above figure 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.

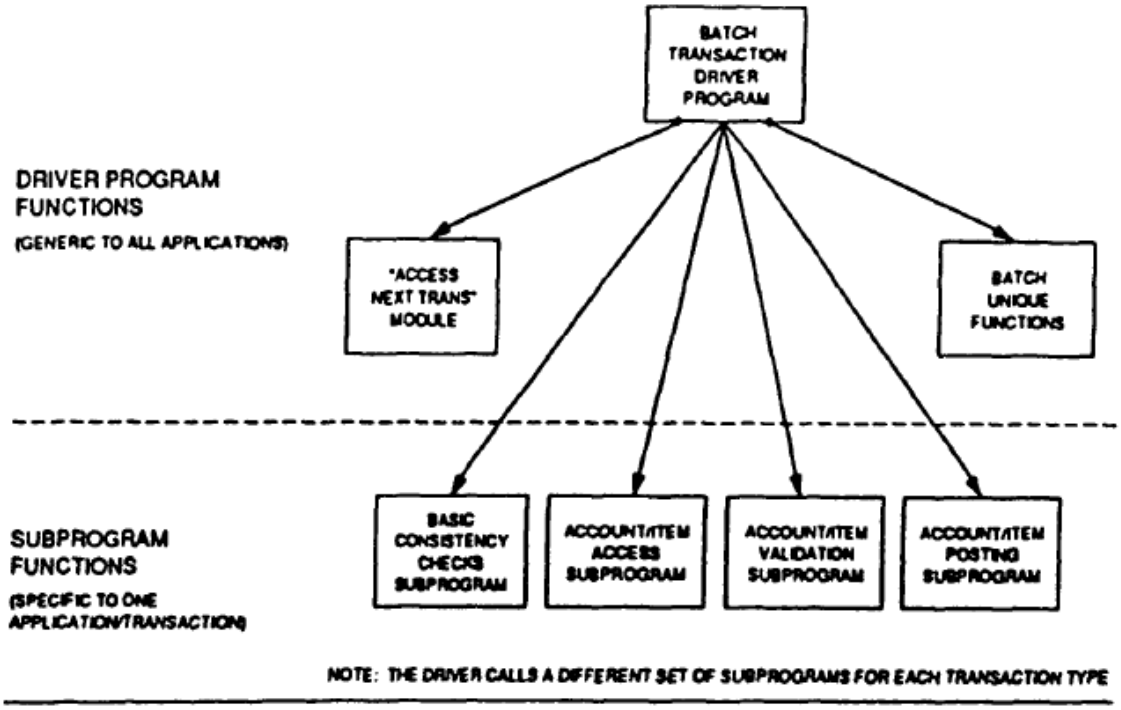


Fig: Internal structure of batch update process

Above figure 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. Here, 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 transaction, and post the result. Thus all the program logic for each transaction is localized in a single set of subprograms.

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.

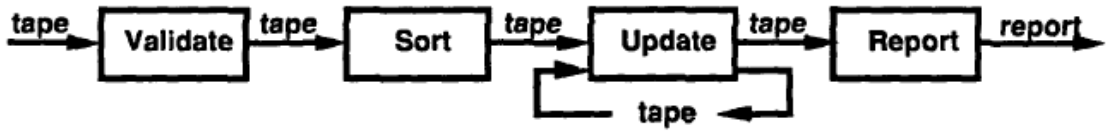


Figure 4: Batch sequential database architecture

Fig: Batch sequential database architecture

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.

 Second, as organizations grew, the set of transactions and queries grew.

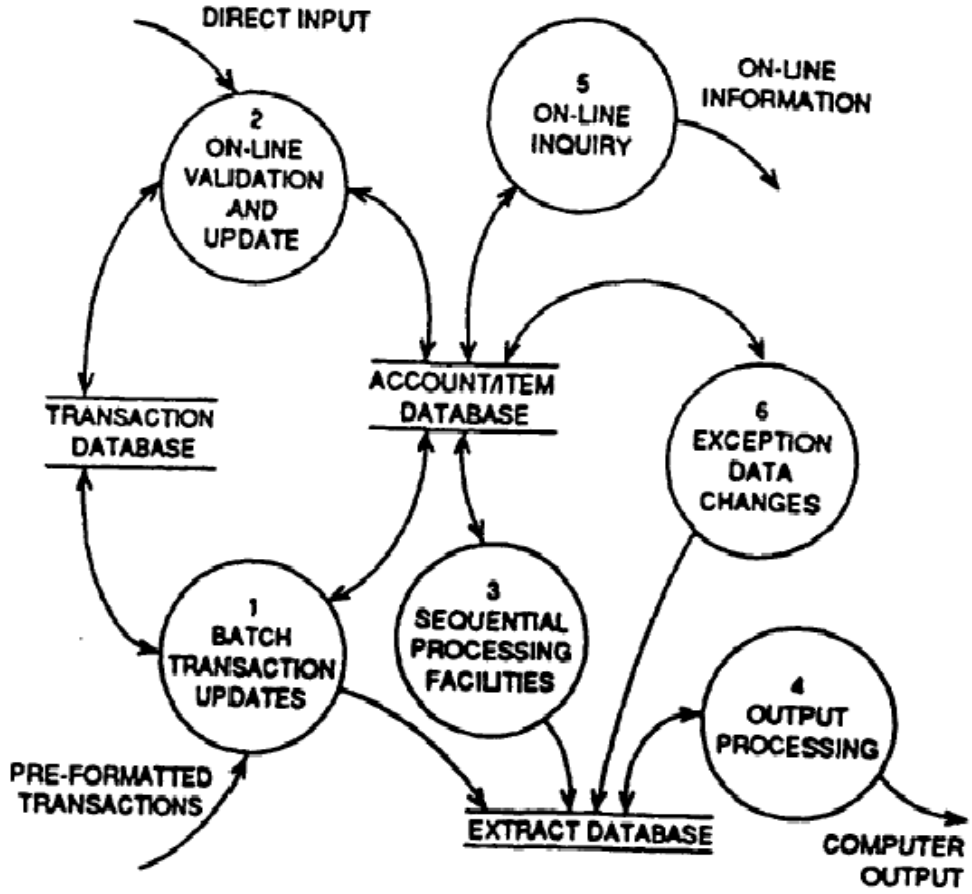


Figure 5: Data fow diagram for interactive database

 Here, 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.

 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.

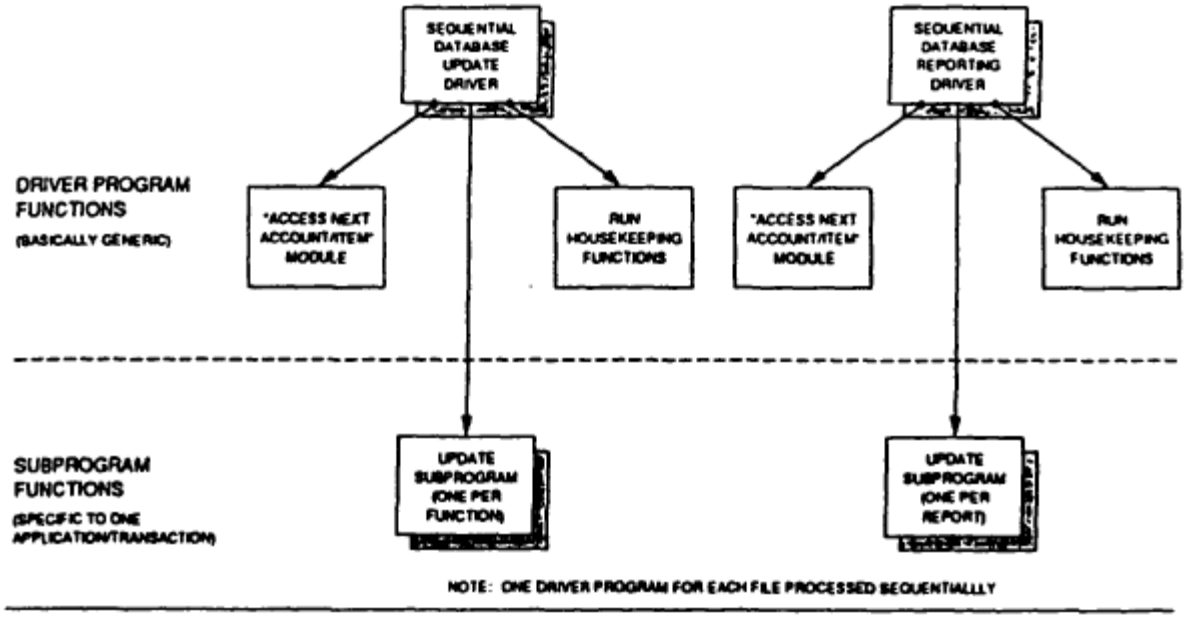


Figure 6: Internal structure of interactive update process

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.

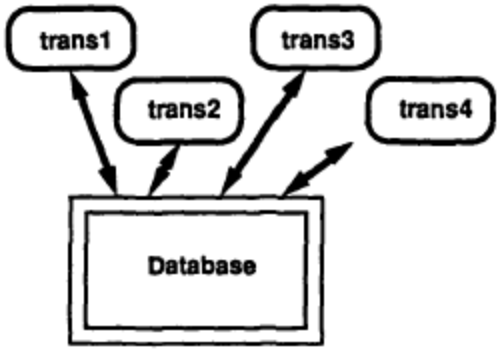


Figure 7: Simple repository database architecture

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.

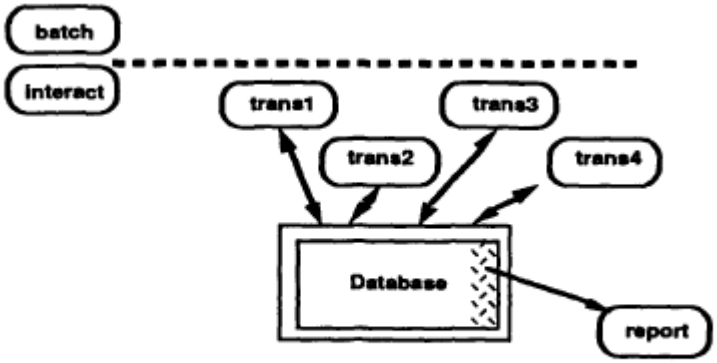


Figure 8: Repository architecture for database showing control and reporting

Virtual Repository:

- As organizations grew, databases came to serve multiple functions.
- 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.
- Developing applications that rely on multiple diverse databases 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.**
- 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.

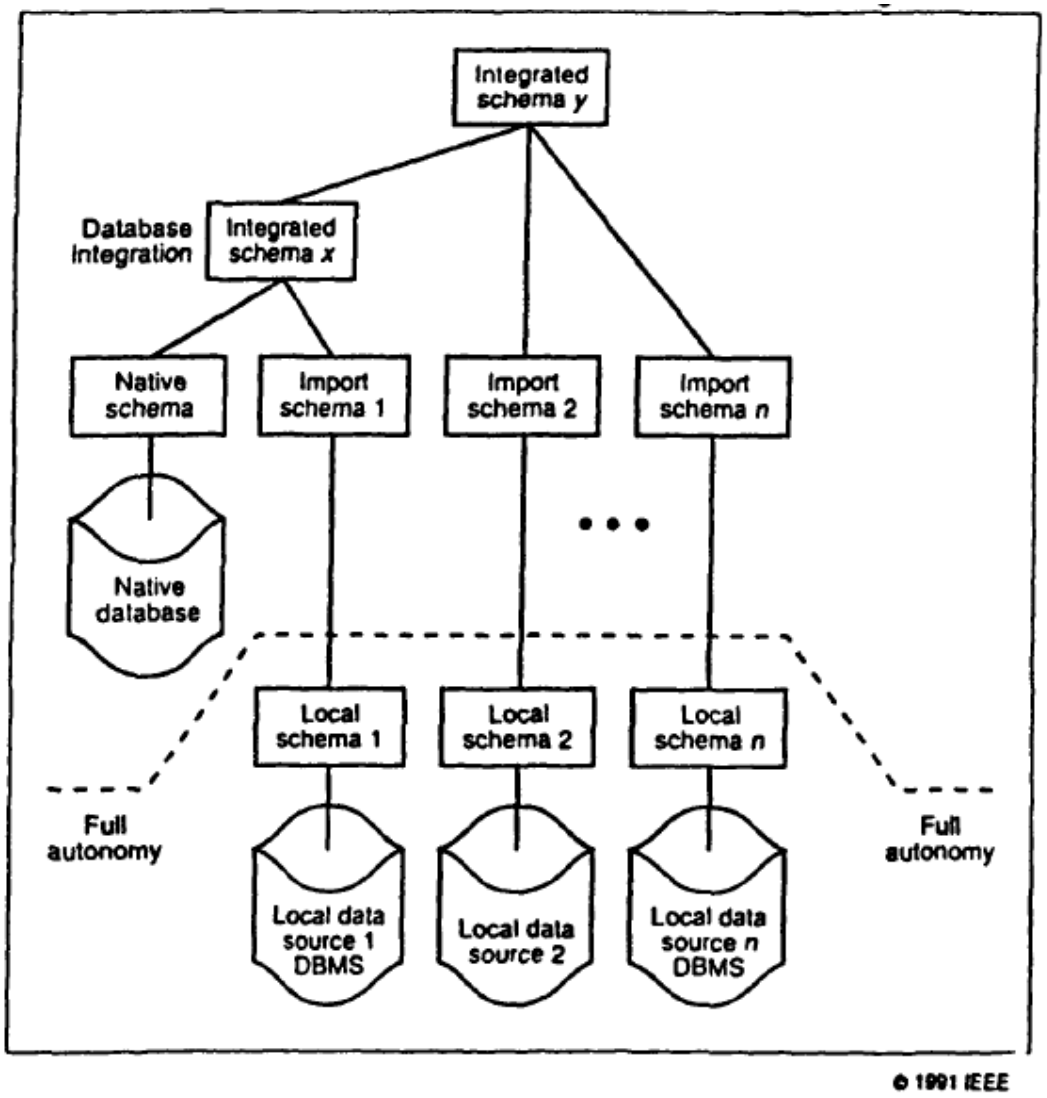


Fig: Combining multiple distributed schemas

- 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;

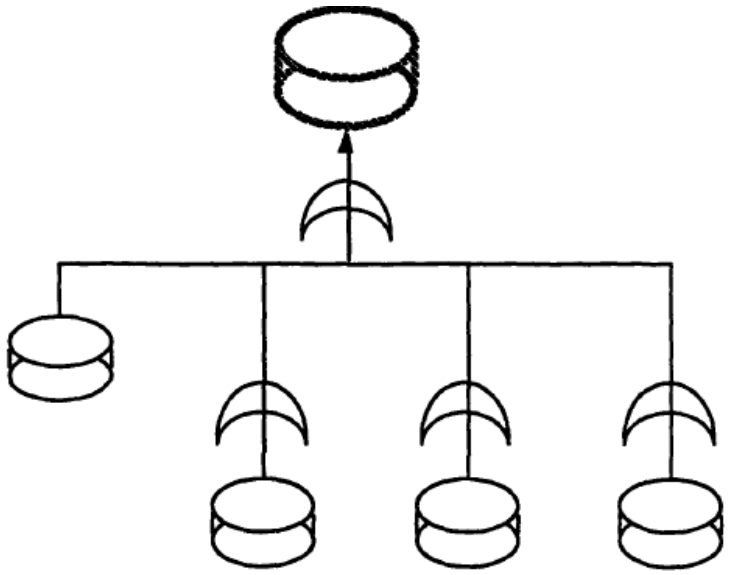


Figure 11: Integration of multiple databases

Hierarchical Layers:

- Unified schemas allow for merger of information, but their mappings are fixed, passive, and static.
- 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.
- 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;
- 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.
- **Problems:** inconsistency across a set of databases, dynamic reconfiguration.

Below fig. 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 user's queries in useful forms, and about how to reconcile, integrate, and interpret information from multiple diverse databases.

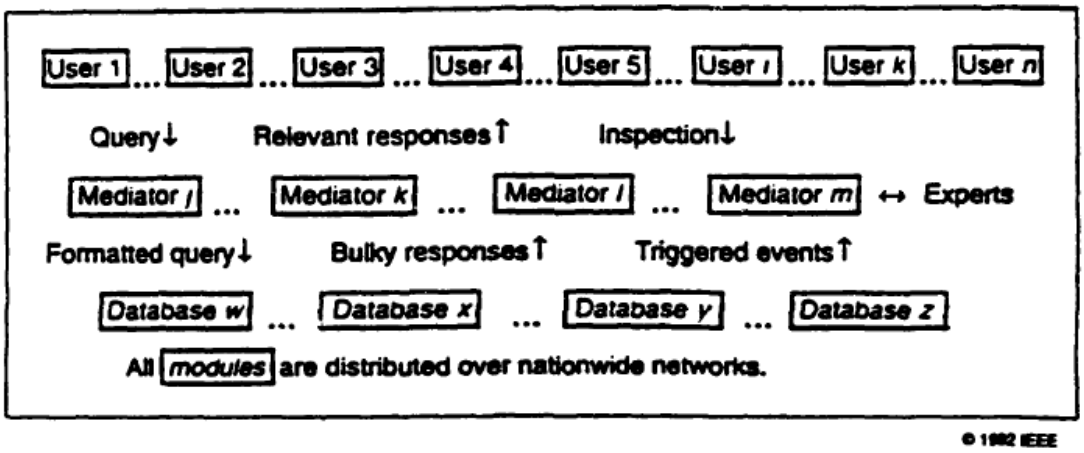


Figure 12: Multidatabase with mediators

Fig: Multi-database with mediator

 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.

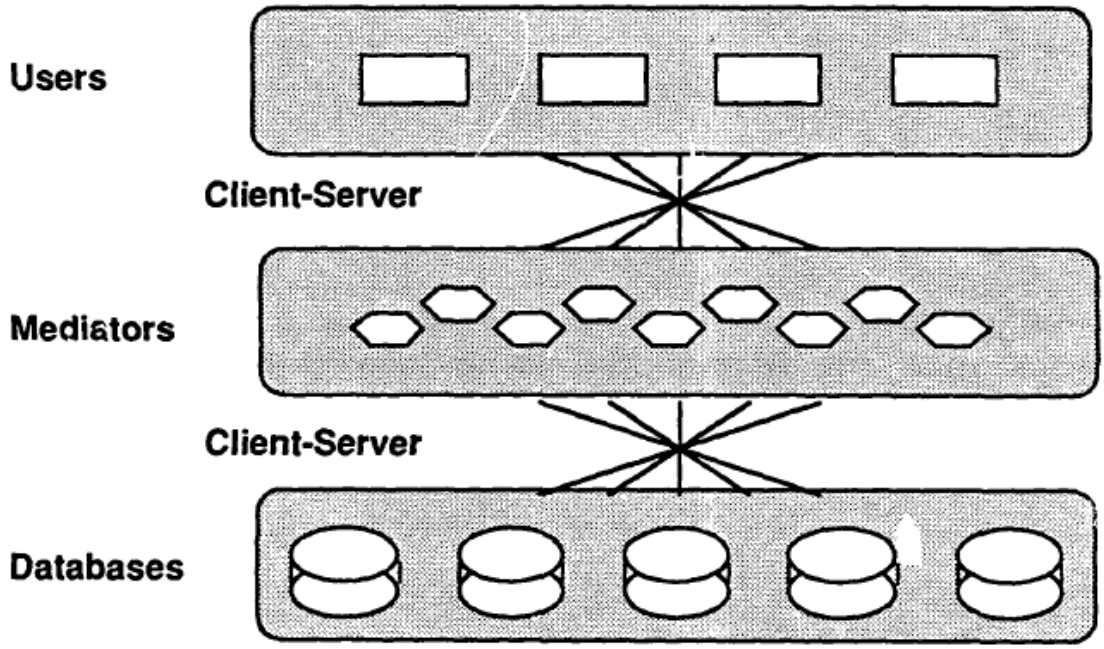


Figure 13: Layered architecture for multidatabase

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.

INTEGRATION IN SOFTWARE DEVELOPMENT ENVIRONMENTS:

 Software development has relied on **software tools** whereas 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.

 Tools now support analysis, configuration control, debugging, testing, and documentation as well.

Batch Sequential:

 The earliest software development tools were standalone programs.

 Often their output appeared only on paper and maybe in the form of object code on cards or paper tape.

 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.

 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.

 Later, 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.

Transition from Batch Sequential to Repository:

 Our view of the architecture of a system can change in response to improvements in technology.

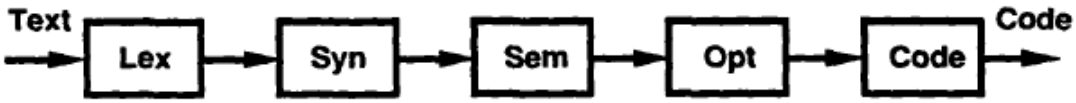


Figure 14: Traditional compiler model

- We often refer to this compilation model as a pipeline, even though it was closer to a batch sequential architecture in which each transformation ("pass") ran to completion before the next one started.
- Symbol Table was not part of the data that flowed from one pass to another but rather existed outside all the passes.
- The algorithms and representations of compilation grew more complex, and increasing attention turned to the intermediate representation of the program during compilation.

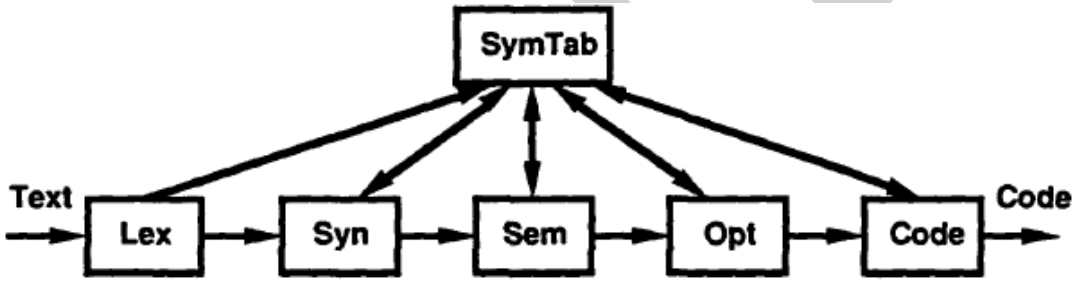


Figure 15: Traditional compiler model with symbol table

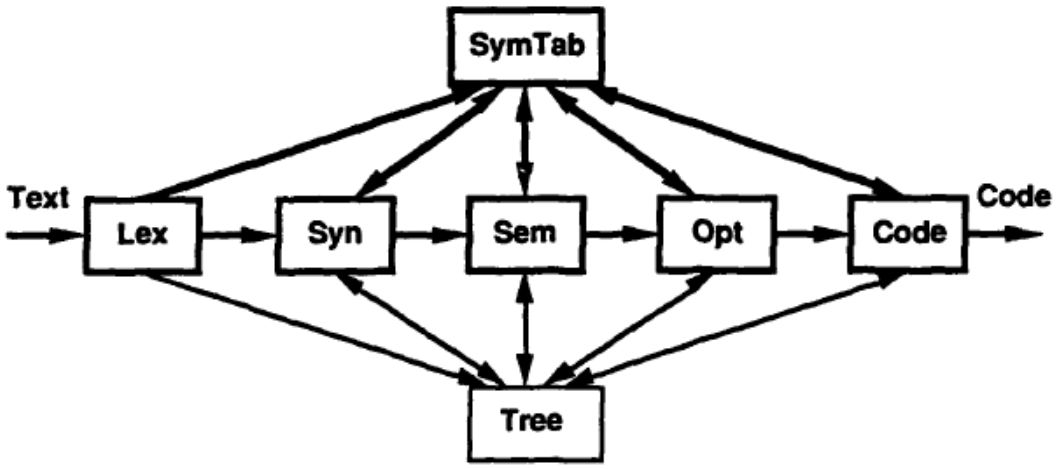


Figure : Modern canonical compiler

 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.

 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*.

 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.

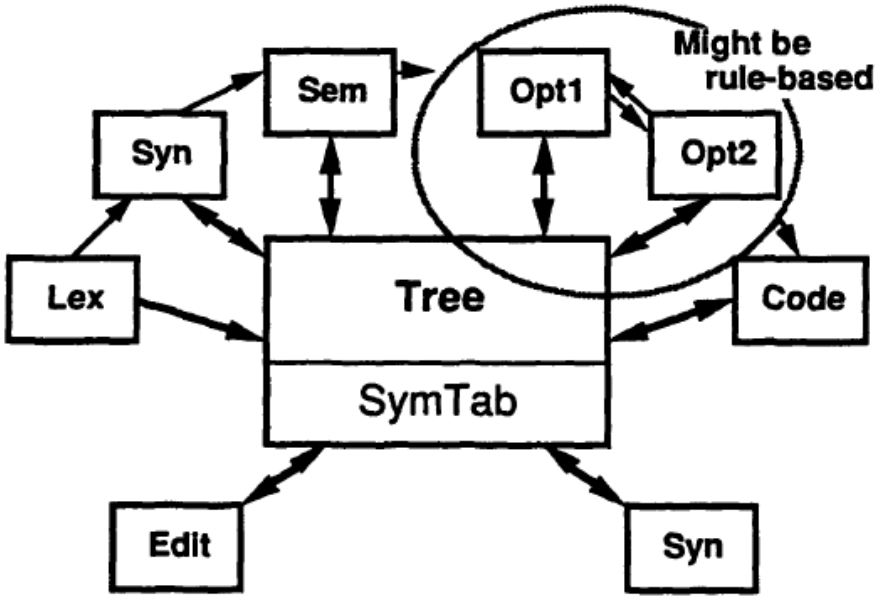


Figure 17: Repository view of modern compiler

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 repository of the compiler provided a focus for this data collection.
- 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.

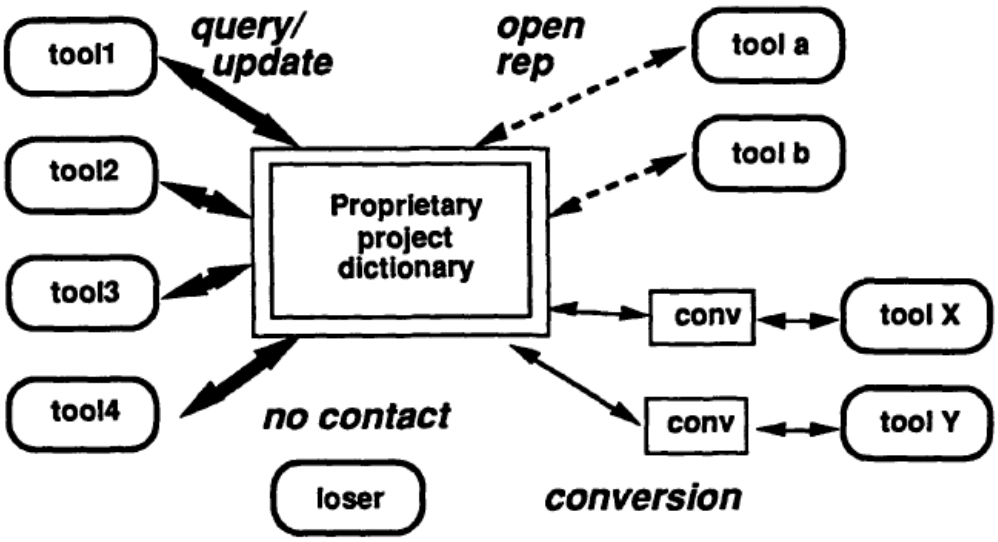


Figure 18: Software tools with shared representation

Hierarchical Layers:

- Current work on integration emphasizes interoperability of tools, especially in distributed systems.
- 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.
- This model provides for integration of data, it provides communication and user interface services directly.
- 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.

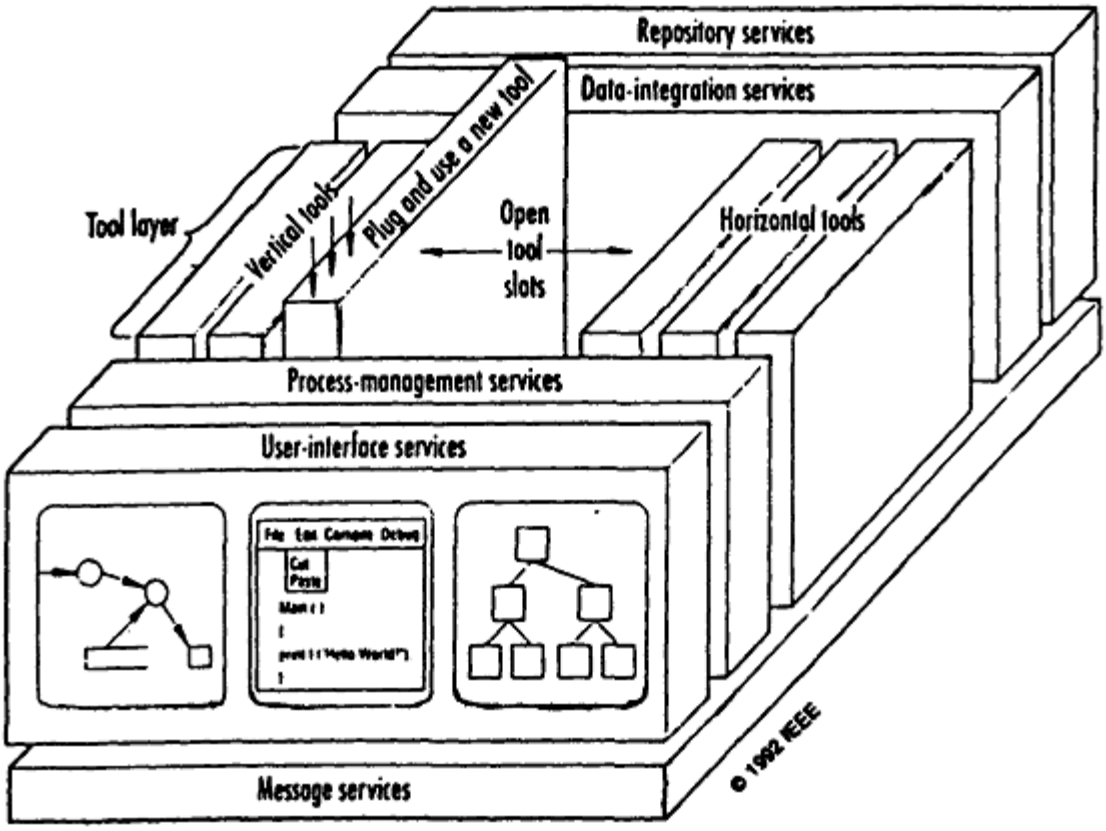


Figure 19: NIST/ECMA reference model for environment integration

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.
- 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 inflexible.
- Software development environments are under pressure to add capabilities for handling complex dependencies and selecting which tools to use.
- 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.
- The use of well-known patterns leads to a kind of reuse of design templates.

Variants on Data Flow Systems:

- The data flow architecture that repeatedly occurs in the evolution of shared information systems is the batch sequential pattern.

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 sub industries 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 pair wise 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 sub problems. 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 sub problems is forced by the need for specialization and the nature of the industry; the problems are not inherently decomposable, and the sub problems 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.

Repository

Selection of tools and composition of individual results requires judgr, nt, 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 sub-problems. 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 sub-problems; the problems are not inherently decomposable, and the sub-problems 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.

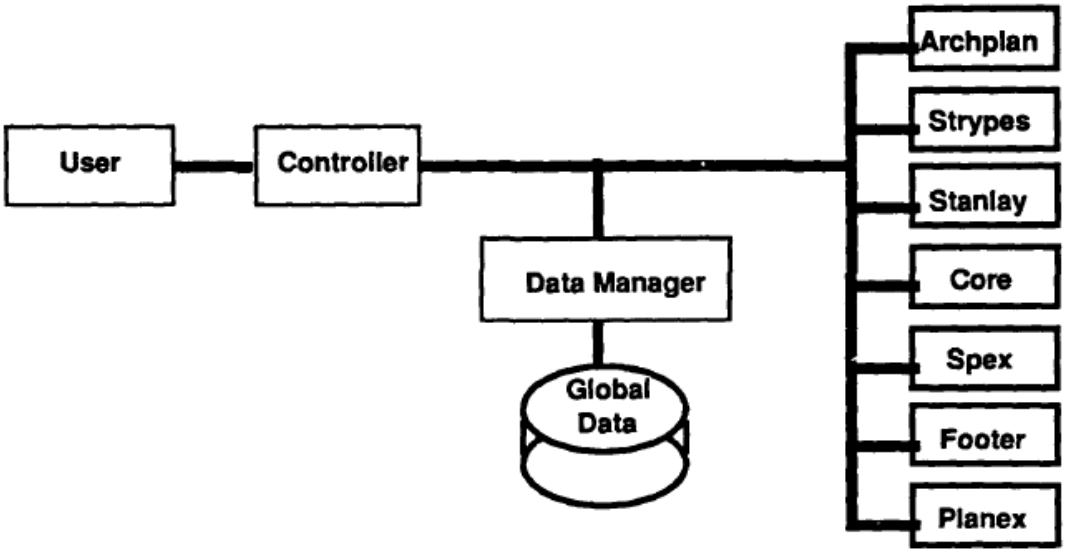


Figure 20: Integrated building design environment

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 sys items 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.

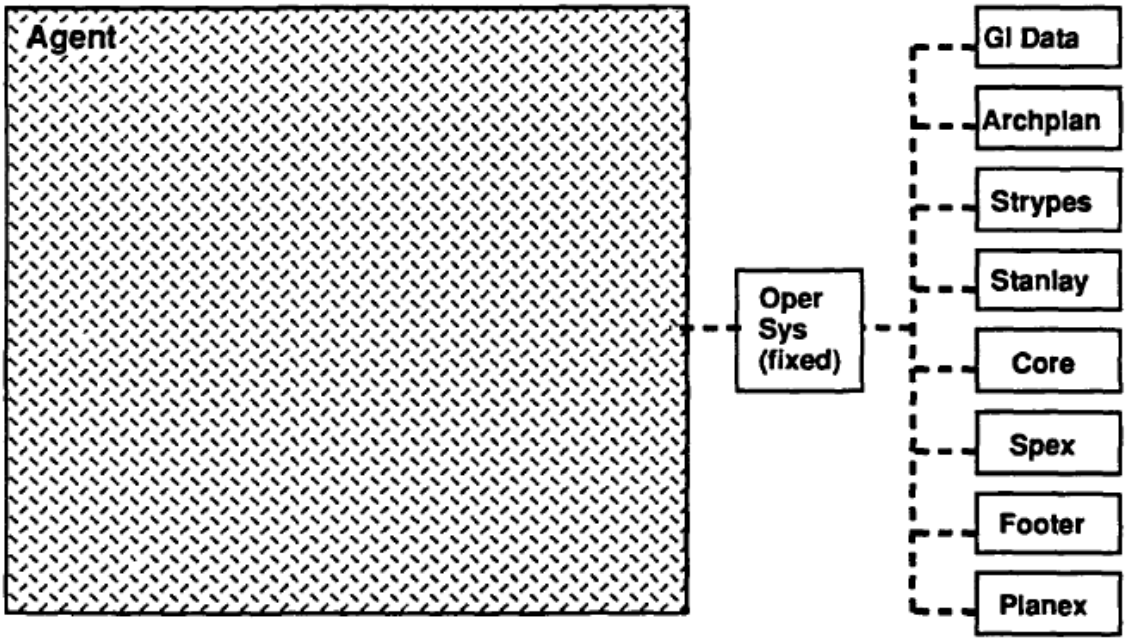


Figure 21: High-level architecture for intelligent IBDE

Figure 22 adds the fine structure of the intelligent agent. The agent has six major components 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.

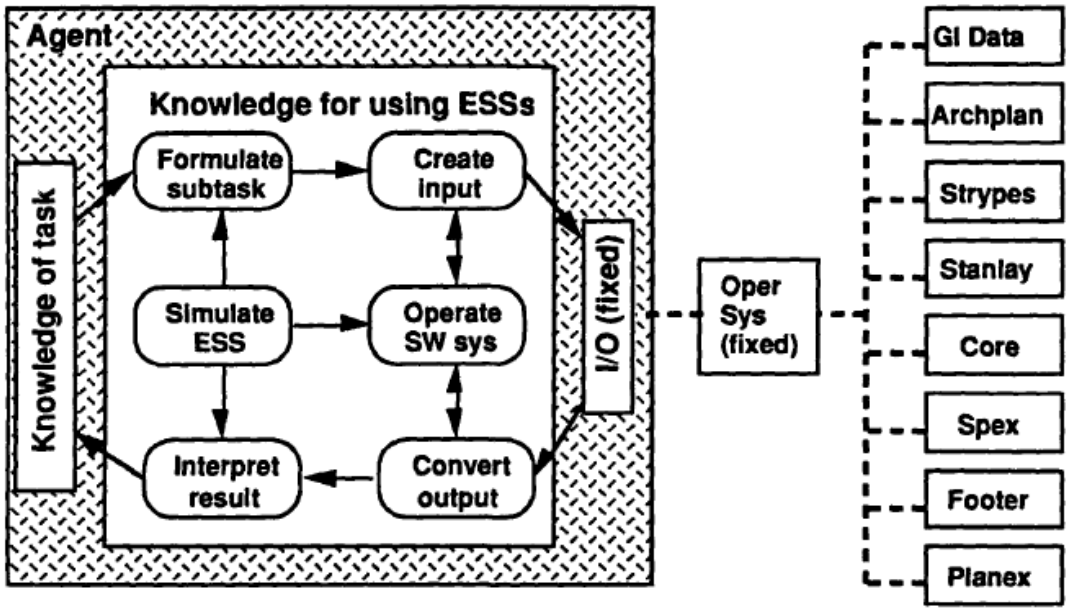


Figure 22: Detailed architecture for Soar/IBDE

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.

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.

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

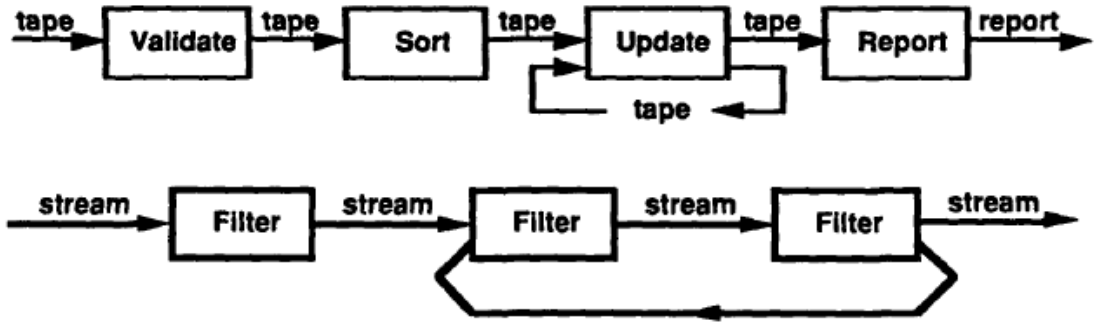


Figure 23 a, b: Comparison of (a) batch sequential and (b) pipe/filter architectures

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 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 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.

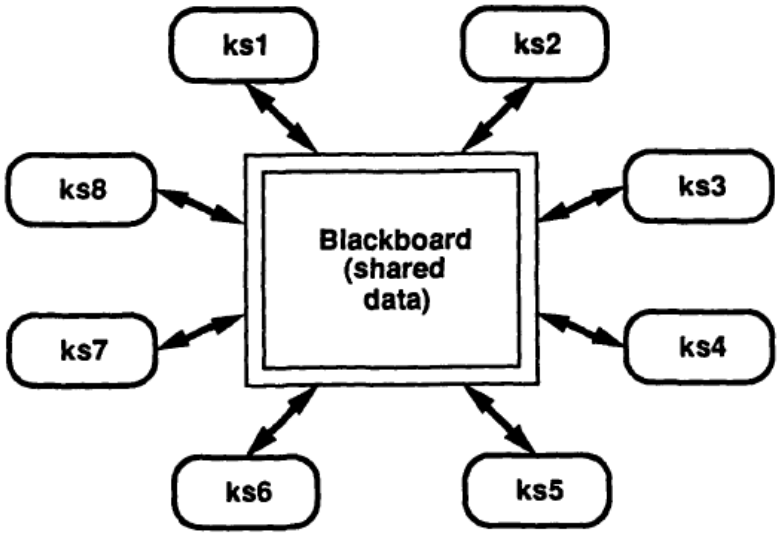


Figure 24: Blackboard architecture

POSSIBLE QUESTIONS

PART – B

1. Write in detail about Database Integration – Simple and Virtual Repository.
2. Elaborate Architectural Structures for Shared Information Systems
3. Explain Database Integration (i) Batch Sequential (ii) Hierarchical Layers
4. Elaborate the Integration in the Design of Buildings an example.
5. Explain in detail about Integration in Software Development Environments
6. Elucidate on the Integration in Building design - Intelligent Control.
7. Explain about Integration in Software Development
8. Write in detail on Architectural Structures for Shared Information Systems - Variants on Dataflow Systems
9. Write in detail about Database Integration Virtual Repository.
10. Write in detail about Evolution of Shared Information Systems in Building Design

PART – C

1. Write in detail about Database Integration
2. Explain in detail about Integration in Software Development Environments
3. Elucidate on the Integration in Building design

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS: III BSC CS SUBJECT NAME: INTRODUCTION TO SOFTWARE ARCHITECTURE
SUBJECT CODE: 17CSP204 BATCH-2017-2019

UNIT: III (ONE MARKS) PART A - ONLINE EXAMINATION

S.NO	QUESTION	CHOICE 1	CHOICE 2	CHOICE 3	CHOICE 4	ANSWER
1	_____ is class of large systems responsible for collecting, manipulating, and preserving large bodies of complex information.	Shared Information Systems	Distributed Information Systems	Central Information Systems	Datastore	Shared Information Systems
2	The earliest shared information systems consisted of separate programs for separate _____	subtasks	main task	components	modules	subtasks
3	_____ is driven primarily by the need to build business decision systems from conventional databases	Business design	Data Processing	Software development	software management	Data Processing
4	When requirements for interaction appear, new organizations allowed independent processing subsystems to interact through_____ .	a shared datastore	pipes	filters	components	a shared datastore
5	Business data processing has traditionally been dominated by database management, in particular by _____	Business data	database updates	money transfer	transactions	database updates
6	_____ driven primarily by the need to represent and manipulate programs and their designs.	software development environments	Database updates	Data updates	Database	software development environments
7	_____ driven couple independent design tools to allow for the interactions of their results in structural design.	Data Processing	Building design	Data updates	Database	Building design
8	A massive_____ which accepted transaction inputs and performed such validation as was possible without access to the database	transaction sort	edit program	update programs	Data Processing	edit program
9	A massive _____ which got the transactions into the same order as the records on the sequential master file	transaction sort	edit program	update programs	Data Processing	transaction sort

10	A sequence of _____executed the transactions by moving sequentially through the master file	transaction sort	update programs	edit program	Data Processing	update programs
11	_____ program produced periodic reports	a print program	update programs	edit program	Data Processing	a print program
12	In Yourdon data flow diagram, processes are depicted as _____	Computer	circles or bubbles	Boxes	arrows	circles or bubbles
13	Data flow is depicted with	arrows	circles, or bubbles	Boxes	Square	arrows
14	Data stores such as computer files are depicted with _____.	arrows	circles, or bubbles	parallel lines	Square	parallel lines
15	Each transaction has a standard _____ that check the transaction request, access the required data, validate the transaction, and post the result	set of subprograms	modules	layers	task tree	set of subprograms
16	In Simple Repository interactive technology provided the opportunity and demand for continuous processing of _____ and _____	on-line updates and on-line Queries	offline updates and offline Queries	on-line updates and offline Queries	on-line analysis and offline Queries	on-line updates and on-line Queries
17	The transaction database serves to synchronize _____	multiple updates	Interactive Database	on-line updates	off-line updates	multiple updates
18	Multiple operations on a single account all at once can be done in	multiple updates	Interactive Database	on-line updates	off-line updates	Interactive Database
19	___ in which shared persistent data is manipulated by independent functions each of which has essentially no permanent state.	a repository	data warehouse	on-line updates	off-line updates	a repository
20	The two additional structures of a simple repository are the _____ and _____	batch and interactive transactions	control element and buffer	update and query operations	add and delete operation	control element and buffer
21	Corporate reorganizations, mergers, and other consolidations of data forced the joint use of _____	multiple databases	Simple Repository	single database	interactive database	multiple databases
22	In _____ database mechanisms integrate multiple schemas into a single schema	Virtual Repository	Simple Repository	Database updates	Database Queries	Virtual Repository

23	For each database devise a schema in its native _____ that exports to the importer.	schema language	database Language	Query Language	Report Language	schema language
24	One approach to the unification of multiple schemas is called the _____ approach	unified	federated	Database	Repository	federated
25	_____ allow for merger of information, but their mappings are fixed, passive, and static	Unified schemas	native schema	the importer	matching schema	Unified schemas
26	_____ simply transform the underlying data.	mappings	schemas	abstraction	database	mappings
27	The experts or _____ localize knowledge about how to discover what databases are available and interact with them	developers	analysts	active mediators	administrators	active mediators
28	_____ are Standalone programs; results are passed from one to another on magtape	Batch processing	Interactive processing	Unified schemas	Multi-database	Batch processing
29	_____ is a Repository model with external control	Batch processing	Interactive processing	Unified schemas	Multi-database	Interactive processing
30	One virtual repository defines (passive) consistent conversion mappings to multiple databases is called _____	Batch processing	Interactive processing	Unified schemas	Multi-database	Unified schemas
31	_____ is a databases have many users, passive mappings don't suffice, active agents mediate interactions	Batch processing	Interactive processing	Unified schemas	Multi-database	Multi-database
32	Larger memories and faster processing enable access to an ever-wider assortment of data resources in a _____	heterogeneous and distributed world	heterogeneous and centralized world	homogeneous and centralized world	homogeneous and distributed world	heterogeneous and distributed world
33	These _____ applications exhibit a pattern of development driven by changing technology and changing needs	business data processing	Batch processing	Interactive processing	Multi-database	business data processing
34	Software development has relied on software tools for almost as long as data processing has relied on _____	on-line databases	data processing	Software development	None of these	on-line databases

35	_____ has relied on software tools for almost as long as data processing has relied on on-line databases.	software tools	Software development	data processing	None of these	Software development
36	Software development has relied on _____ for almost as long as data processing.	data processing	Software development	software tools	None of these	software tools
37	_____ does translation from source code to object code	compiler	assembler	libraries	interpreter	compiler
38	_____ support analysis, configuration control, debugging, testing, and documentation as well.	software tools	Software development process	configuration control	software engineering	software tools
39	The earliest software development tools were _____programs.	standalone	object code	Scripts	pipeline	standalone
40	Most compilers created a separate _____ during lexical analysis and used or updated it during subsequent passes.	Symbol Table	Data Flow	Memory	Computations	Symbol Table
41	The intermediate representation for example, _____ was the center of attention.	an attributed parse tree	Symbol Table	Data Flow	Computations	an attributed parse tree
42	A more appropriate view of _____ structure would re-direct attention from the sequence of passes to the central shared representation.	Traditional compiler	Repository view of compiler	Modern canonical compiler	hierarchical view	Modern canonical compiler
43	_____ of Modern Compiler accommodates various tools that operate on the internal representation rather than the textual form of a program.	Repository View	Traditional compiler	Modern canonical compiler	hierarchical view	Repository View
44	_____ Share detailed knowledge of the common, but proprietary, representation among the tools of a single vendor	Tight coupling	Loosely Coupling	Conversion	representation	Tight coupling
45	_____ Publish the representation so that tools can be developed by many sources	Tight coupling	Loosely Coupling	Open representation	Conversion boxes	Open representation

46	_____ Provide filters that import or export the data in foreign representations	Tight coupling	Conversion boxes	Open representation	No contact	Conversion boxes
47	_____ Prevent a tool from using the repository, either explicitly, through excess complexity, or through frequent changes.	Tight coupling	Conversion boxes	Open representation	No contact	No contact
48	_____ model provides for integration of data, it provides communication and user interface services directly	Hierarchical Layers	Repository Services	Process Management	Message services	Hierarchical Layers
49	One variation on the integrated-environment theme, the integration system defined a set of _____ and provides support for tools to announce the occurrence of events.	Modules	events	connecters	Components	events
50	The advent of _____ which drove the shift from batch to interactive processing for many functions.	on-line computing,	incremental development	compilation	Integration	on-line computing,
51	The construction community operates on divide-and-conquer problem solving with interactions among the _____	incremental development	subproblems	compilation	Integration	subproblems
52	Integrated environments for building design are frameworks for controlling a collection of _____ that solve part of the building design problem	standalone applications	application software	real time software	system software	standalone applications
53	The goal of an _____ is integration of data, design decisions, and knowledge.	integrated environment	hierarchical view	repository view	traditional view	integrated environment
54	_____ divide into tasks for individual agents	Problem partitioning	Task distribution	Agent control	Agent communication	Problem partitioning
55	_____ is exchange of information essential when subtasks interact or conflict	Agent communication	Data	Communication	Control	Agent communication

56	_____ assign tasks to agents for best performance	Problem partitioning	Task distribution	Agent control	Agent communication	Task distribution
57	_____ is strategy that assures tasks are performed in organized fashion	Problem partitioning	Task distribution	Agent control	Agent communication	Agent control
58	_____ is a split between closed, tools specifically built for this system and open, external tools can be integrated	Tools	Data	Communication	Agent communication	Tools
59	_____ is mostly single-level hierarchy; tools at bottom; coordination at top	Tools	Data	Control	Agent communication	Control
60	_____ is mostly fixed partitioning of kind and processing order; scripts sometimes permit limited flexibility	Tools	Planning	Control	Agent communication	Planning

UNIT-IV

Guidance for User-Interface Architectures – The quantified Design Space – The value of Architectural formalism – Formalizing the Architecture of a specific system – Formalizing an Architectural Style – Formalizing an Architectural Design Space – Towards a Theory of Software Architecture – Z Notation

GUIDANCE FOR USER INTERFACE ARCHITECTURES

The architectural alternatives available to a system designer can be described and classified by constructing a *design space.* Within a design space, we can formulate design rules that indicate good and bad combinations of choices. Such rules can be used to select an appropriate system design based on functional requirements. The design space is useful in its own right as a shared vocabulary for describing and understanding systems.

This work should be viewed as a means of codifying software design knowledge for use in day-to-day practice and in the training of new software engineers. For this purpose, a set of design rules need not produce a "perfect" or "best possible" design. A valuable contribution will be made if the rules can help a journeyman designer to make choices comparable to those that a master designer would make—or even just help the journeyman to choose a reasonable design with no major errors. With sufficient experience, a set of such rules may become complete and reliable enough to serve as the basis for automated system design, but the rules can be of practical use long before that stage is reached.

The work described in this report tested these notions by constructing a design space and rules for the architecture of user interface software systems. These rules were experimentally tested by comparing their recommendations to actual system designs. The results showed that a rather simple set of rules could achieve a promising degree of agreement with the choices of expert designers. These exploratory results suggest that the approach sketched here is a viable means of creating an organized body of knowledge for software engineering.

This report is a summary of results from the author's thesis [Lane 90a]. A companion report presents the user interface design space and rules in greater detail

1. The Utility of Codified Knowledge

The underlying goal of this work is to organize and express software design knowledge in a useful form. One way of doing this is to build up a vocabulary of well-understood, reusable design concepts and patterns. If widely adopted, a design vocabulary has three major benefits. First, it aids in creating a system design by providing mental building blocks. Second, it helps in understanding or predicting the properties of a design by offering a context for the creation and application of knowledge. Third, it reduces the effort needed to understand another person's design by reducing the number of new concepts to be learned.

An example of such a vocabulary is the codification of control structures that took place about two decades ago. Programmers learned to perceive control flow in terms of a few standard concepts (conditionals, iteration, selection, subroutine calls, etc.) rather than as a complex pattern of low-level tests and branches. By reducing apparent complexity and providing a shared understanding of control flow patterns, use of these building blocks made programs both easier to write and easier to read. Researchers discovered key properties of these structures, for example, the invariant and termination conditions of loops. Use of the standard structures helped practitioners to focus on these properties, leading to better-understood, more reliable programs. Finally, codification made it possible to build tools (programming languages) that supported the structural concepts directly, providing further productivity gains.

As software engineering matures and research attention shifts to ever-larger problems, we can expect to see similar codification occurring for larger software entities. The time now seems ripe to begin codifying structural patterns in medium-size software systems, to witness characteristics of modules and the interconnections between them.

(We can already anticipate that even higher levels of design abstraction will be needed to design very large systems, but we are far from having enough experience to be able to discern patterns at that scale.) A different analogy for this wok is the compilation of engineering design handbooks, such as [Perry 84]. The established fields of engineering have long distinguished between innovative and routine design. Innovative design relies upon raw invention or derivation from abstract principles, while routine design uses standardized methods to solve problems similar to those that have been solved before. When applicable, routine design methods are cheaper and more likely to yield an acceptable (though not necessarily optimum) design than are innovative methods. The primary purpose of such handbooks is 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. Thus, software engineering handbooks could

combat two opposite evils now widely seen in practice: both the tendency to invent every new system from scratch and the tendency to reuse a single design for every problem regardless of its suitability. Handbook-style texts are now widely available for selection of algorithms and data structures (e.g., [Knuth 73, Sedgewick 88]) but do not yet exist for higher levels of software design.

The work reported here offers an organizational scheme (namely, design spaces and mles) for handbooks of software system structure, as well as the beginnings of specific knowledge for one such handbook (covering user interface systems).

1.1 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.

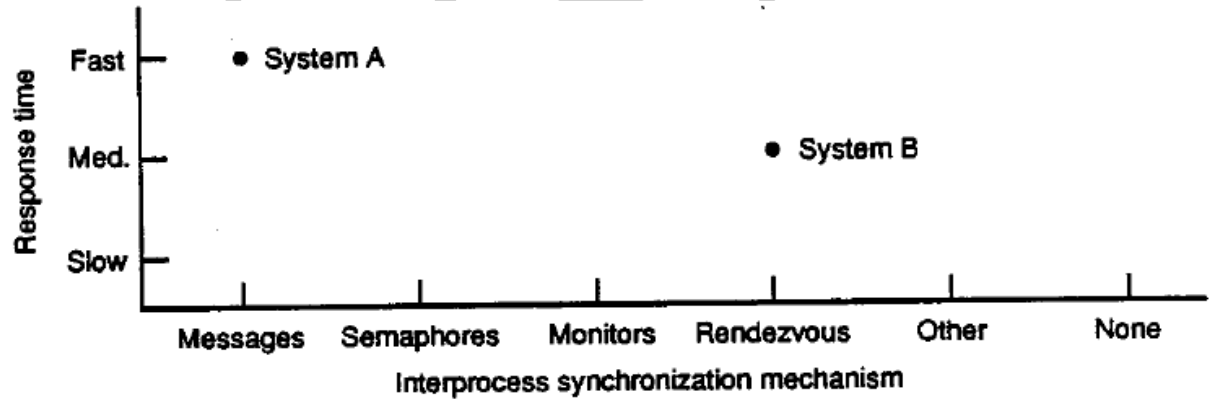


Figure 1-1: A Simple Design Space

The different dimensions are not necessarily independent; in fact, it is important to discover correlations between 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 key part of die design ϕace approach is to choose some dimensions that reflect requirements or evaluation criteria (function and/or performance), while other dimensions reflect structure (or other available design choices).

Then, any correlations found between these dimensions can provide direct design guidance: they show which design choices are most likely to meet the functional requirements for a new system. For example, the hypothetical data in Figure 1-1 suggest that a message mechanism is more likely to provide fast response time than a rendezvous mechanism. (Of course, one would want more than just two data points before drawing this conclusion.)

The dimensions that describe functional and performance requirements make up the functional design space, while those that describe structural choices make up the structural design space. These groupings can be regarded either as independent spaces or as subspaces of a single large design space. In the context of a stepwise ("waterfall") model of the software design process, the functional design space represents the results of the requirements analysis and gross functional design steps, while the structural design space represents the results of initial system decomposition.

The dimensions of a design space are usually not continuous and need not possess any useful metric (distance measure). A dimension that represents a structural choice is likely to have a discrete set of possible values, which may or may not have any meaningful ordering. 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 (e.g., a performance number), one may choose to aggregate it into a few discrete values (e.g., "low," "medium," "high"). This is appropriate when such gross estimates provide as much information as one needs or can get, as is often true in the early stages of design.

2. A Design Space for User Interface Architectures

The design space reported here, together with its associated rules, describes architectural alternatives for user interface software: systems whose main focus is on providing an interactive user interface for some software functions). The system studied need not provide the whole user interface. Thus the scope of the study included not only complete user interface management systems (UIMSs), but also graphics packages, user interface toolkits, window managers, and even standalone applications that have a large user interface component This scope is large enough to include a wide range of useful system structures, yet not so large as to be intractable. While another domain could have been chosen, user interfaces are a good choice because the field is in ferment, with little agreement on the best possible structures. Hence the results may be useful immediately, in addition to serving to illustrate the larger argument made above.

The design space is too large to cover completely in this report Therefore only some representative dimensions and rules will be described. (For a more complete presentation of the space, see [Lane 90b].) The complete design space contains 25 functional dimensions, 6 of which are described here. Three to five alternatives are recognized in each of these dimensions. There are 19 structural dimensions (5 of which are described here), each offering two to seven alternatives. Figure 2-2 presents the dimensions discussed in this report

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 This consists of 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 This consists of 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 This consists of 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).

The inter module 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

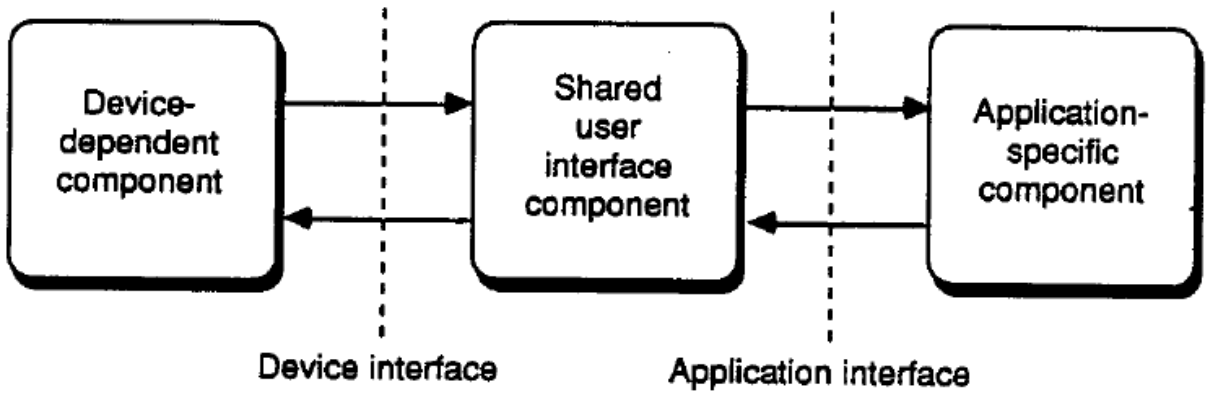


Figure 2-1: A Basic Structural Model for User Interface Software

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, regarding the server as a device-specific black box.

Sample Functional Dimensions

The functional dimensions identify the requirements for a user interface system that most affect its structure.

These dimensions fall into three groups:

• **External requirements** This group 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.** This group includes the key decisions about user interface behavior that fundamentally influence internal structure.

• **Practical considerations.** This group covers development cost considerations; primarily, the required degree of adaptability of the system.

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.

External Requirements

External event handling is an example of a dimension reflecting an application-imposed external requirement.

This dimension indicates whether the application program needs to respond to external events (defined as events not originating in the user interface), and if so, on what time scale. The design space recognizes three alternative choices:

• **No external events:** the application is not influenced 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 interlace.

• **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. Automatic reporting of mail arrival might be handled this way.

• **External events** preempt user commands: external event servicing has sufficiently high priority that user command execution must be interrupted when an external event occurs. This requirement is common in real-time control systems.

User customizability is an example of a user-imposed external requirement. The design space recognizes three levels of end user customizability of a user interface:

• **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, window sizes, colors, etc.

• **Low:** little or no user customizability is required.

Functional Dimensions	**Structural Dimensions**
External event handling • No external events • Process events while waiting for input • External events preempt user commands	Application interface abstraction level • Monolithic program • Abstract device • Toolkit • Interaction manager with fixed data types • Interaction manager with extensible data types

User customizability • High • Medium • Low User interface adaptability across devices • None • Local behavior changes • Global behavior changes • Application semantics changes Computer system organization • Uniprocessing • Multiprocessing • Distributed processing Basic interface class • Menu selection • Form filling • Command language • Natural language • Direct manipulation Application portability across user interface styles • High • Medium • Low	• Extensible interaction manager Abstract device variability • Ideal device • Parameterized device • Device with variable operations • Ad-hoc device Notation for user interface definition • Implicit in shared user interface code • Implicit in application code • External declarative notation • External procedural notation • Internal declarative notation • Internal procedural notation Basis of communication • Events • Pure state • State with hints • State plus events Control thread mechanism • None • Standard processes • Lightweight processes • Non-preemptive processes • Event handlers • Interrupt service routines

Figure 2-2: The Sample Design Space Dimensions

User interface adaptability across devices depends on the expected range of I/O devices that the user interlace system must support This dimension indicates the extent of change in user interface behavior that 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 occur across devices, for example, in the appearance of menus.

• **Global behavior changes:** there are major changes in surface user interface behavior across devices, for example, a change in basic interface class (see below).

• **Application semantics changes:** there are changes in underlying semantics of commands (e.g.,

continuous display of state versus display on command).

Computer system organization is an example of a dimension describing the surrounding computer system. This dimension classifies the basic nature of the environment as follows:

• **Uniprocessing:** only one application executes at a time.

• **Multiprocessing**: multiple applications execute concurrently.

• **Distributed processing:** environment is a computer network, with multiple CPUs and non-negligible communication costs.

Basic Interactive Behavior

Bask interface class identifies the basic kind of interaction supported by the user interface system. (A general purpose system might support more than one of these classes.) Hie design space uses a classification :

• **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. Resolution of ambiguous input is a key problem.

• **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.

Practical Considerations

Application portability across user interface styles is an example of a dimension defining the required degree of adaptability of a user interface system. This dimension specifies the degree to which application-specific code is insulated from user interface style changes.

• 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.

Sample Structural Dimensions

The structural dimensions represent the decisions determining the overall structure of a user interface system.

These dimensions also fall into three major groups:

• **Division of functions and knowledge between modules.** This group considers how system functions are divided into modules, the interfaces between modules, and the information contained within each module.

• **Representation issues.** This group considers 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 dialogue window), or only implicitly.

• **Control flow, communication, and synchronization issues.** This group considers the dynamic behavior of the user interface code.

Division of Functions and Knowledge Between Modules

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:

• **Monolithic program:** there is no separation between application-specific and shared code, hence no such 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.

• **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:** similar to the previous category, except that 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:** again, communication between the application and shared code is 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 there is a significant body of application-

specific code that customizes the interaction manager, this code is much more tightly coupled to the internal details of the interaction manager than is the case for clients of non extensibie 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 are 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. Examples are GKS logical input devices 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, an abstract device with variable operations is useful when much of the system's "intelligence" is to be put into the device-specific layer but it is only appropriate for handling local changes in user interface behavior across devices.

Representation Issues

Notation for user interface definition is a representation dimension. It 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 non-procedural specification separate from the body of the application program, for example, a grammar or tabular specification. External declarative notations are particularly well suited for supporting user customization and for use by non-programming 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 definition 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 non-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.

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 notation is the easiest to use (especially for non-programming user interface

designers) but it has the least power, since it can only represent a predetermined range of possibilities. Typically, several notational techniques are used in a system, with different aspects of the user interface being 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.

Design Rules for User Interface Architecture

There are very few hard-and-fast rules at this level of design. Most connections between design dimensions are better described by saying that a given choice along one dimension favors or disfavors particular choices along another dimension; the strength of this correlation varies from case to case. The designer's task is to consider all such correlations and to select the alternative favored by the preponderance of the evidence.

Therefore, a natural notation for a design rule is a positive or negative weight associated with particular combinations of alternatives from two (or more) dimensions. A given design can be evaluated by summing the weights of all applicable rules. The "best" design is then the one with the highest score. The author prepared a mechanically evaluatable set of design rules of this form and an evaluation program that would rank the structural alternatives when given a set of values for the functional dimensions. The rules can also be viewed less formally as guidelines for human designers.

It is useful to distinguish two categories of rules: those linking functional to structural dimensions, and those interconnecting structural dimensions. The first group allows system requirements to drive a structural design, while the second group ensures the internal consistency of the design.3 This second group complicates the task of finding the design with the highest scene, since choices in different dimensions affect each other. The author resorted to combinatorial searching to locate the best designs; better algorithms may be found in the future. A possible source of better methods is "neural network" techniques, which seem to have some similarity to this problem.

The mechanical design rule set contains 622 rules; these rules are written in a very primitive notation and can be reduced to about 170 rules at a more reasonable level of abstraction. The very abbreviated descriptions below account for about ten percent of the formal rules.

Sample Rules

The earlier descriptions of structural alternatives already mentioned some of the conditions under which one alternative may be preferred to another. This section presents more formally some of the specific design rules that connect the sample dimensions. Each of the sample rules is given in prose form, together with a brief justification.

• If external event handling requires preemption of user commands, then 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.

• 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 external notations.

• 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.

• A distributed system organization favors event-based communication. State-based communication requires shared memory or some equivalent, which is often expensive to access in such an environment

• The basic user interface class affects the best choice of application interface abstraction level. For example, menu selection and form filling user interfaces are well served by toolkits and nonextensible interaction managers. But experience has shown that nonextensible interaction

managers are not adequate for direct manipulation, because they don't handle semantic feedback well. Extensible interaction managers and toolkits are the favored alternatives for direct manipulation.

• 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. 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.

Applying the Design Space: An Example

To illustrate these ideas, this section presents a concrete example. The sample system is the cT programming language and environment [Sherwood 88]. It is designed for the creation of high-quality, interactive educational applications, for example, physics simulations or instruction in musical notation. It must be usable by authors who are experts in their particular subject matter, but who have only limited programming experience. Implementations exist on a variety of personal computers and workstations, and portability of application programs across these platforms is an important goal.

Its functional requirements can be described in the terms of the design space. For the sample dimensions previously cited:

• There is no requirement for external event handling; it's not needed in the target class of applications.

• Little or no end user customizability is needed.

• User interface adaptability across devices may require local behavior changes, for instance to fill areas with different patterns when color is not available. The range of supported platforms is not so wide that global behavior changes might be necessary.

• Computer system organization may be uniprocessing or multiprocessing. It does not make special provisions for distributed systems.

• Basic interface class is usually direct manipulation, but menu selection is also used. Each application determines its basic interactive behavior.

• Medium portability of applications across user interface styles is required. In such things as menu appearance, it follows the conventions of the host platform, and the application should be independent of such details.

To describe it structurally, we classify the it programming system itself as the shared user interface code, instructional programs written in it as application-specific code, and the underlying platform (including graphics packages, etc.) as device-specific code. (Notice that this division is already implicit in the functional classification above.)

The architecture of it can then be classified in the sample structural dimensions as follows:

• The application interface abstraction level falls in the toolkit class. Toolkit elements are provided for common constructs such as menus or scrolling text boxes. cTs toolbox is particularly strong in the analysis of text input (recognition of misspelled words, equivalent forms of algebraic expressions, etc).

For other interactive behavior the application resorts to manipulation of the underlying abstract device.

• The device interface uses a parameterized abstract device. Decisions such as how to scale displays to fit the available hardware are handled largely by the shared user interface code (but the application can set policy, such as whether to preserve aspect ratio).

• User interface notation is mostly implicit; some aspects are implicit in the shared code while others are implicit in the application. Limited use is made of internal procedural notation, and there are some toolbox parameters that qualify as internal declarative notation.

• Communication is based on events; no shared state variables are used.

• cT uses basically a single thread of execution. An exception occurs in the development environment:

while editing a cT program, incremental recompilation is done while waiting for user input The

"background" control thread used for this purpose is implemented with an event handler mechanism.

The mechanical rule set is largely able to replicate these design decisions. For example, the rules recommend implicit and internal-procedural user interface notations, because the requirements for user customizability and application portability are not high enough to justify the extra cost of external or declarative notations. The rules recommend strict single-thread control flow, so they disagree on the last of the sample dimensions. This is unsurprising since the decision to provide background recompilation is outside the scope of the present design space

QUANTIFIED DESIGN PROCESS

- **The Quantified Design Space**
  - Overview
  - The dilemma: Formal specification languages allow designers to make assertions about a design but there is still no tool for systematic and quantitative analysis of designs."
  - "A key part of the design space approach is to choose some dimensions that reflect requirements or evaluation criteria (function and/or performance), while other dimensions reflect structure (or other available design choices).

Then, any correlations found between these dimensions can provide direct design guidance: they show which design choices are most likely to meet the functional requirements for a new system."

- Background
  - The QFD is based on two concepts, the design space and the quality function deployment.
  - Design Space Dimensions:
    - A multi-dimensional design space classifies a system's architecture.
    - 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.
      - I.e. response times,
      - means of inter-process 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.

- Design Space Rules :
  - Be careful.. Weight assignments are subjective and to assign subjective measures to subjective rules can have serious consequences. I have found it better to use some information theoretical approach to combine the evidence for a given architecture using the taxonomy of the design space rules using something like Dempster-Shafer (See Shortliffe Paper).
  - This allows us to assemble both positive and negative influences for an architecture in the same taxonomy.
  - We could also use the approach suggested by Wolfgang Spohn in ranking the plausibility (actually the implausibility) of one architecture alternative over another.

- Rules must be broken down into relationships (define the correlations) between alternatives of different dimensions.

- QFD: "The voice of the customer must be maintained at each step of the product development process"
    - Use the use case description form and approach I gave you as a start to capture the design issues, notes, and assumptions.

The description of the operation of the architecture through the use case will be constrained by the physical configuration of the end-item product.

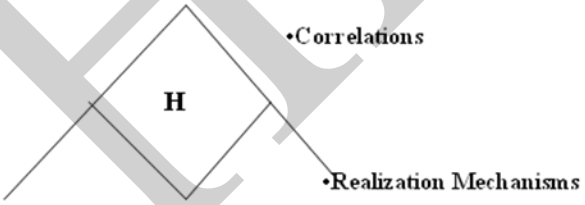
- QFD Process: (We will compare the book's approach against recent published papers)
    - Establish the scope of the architecture being developed (Rows):
        - not as easy as it sounds especially if your system has to interoperate with another system that requires modification to work with yours.
        - Include the Quality Attributes !!!!, not just functionality !!!
    - Identify the (Potential) Realization Mechanisms (Columns)
        - Create a matrix of customer requirements to realization mechanisms : (See book example)
    - Establish Target Values for each realization mechanism
        - Note the need to be Objectively Verifiable!
        - Not their location on the Matrix.
        - Design Space rules can be captured in these values
    - Establish the Relationship (row/column correlation strength) between each (potential) mechanism and the customer need.
        - (Identify) the mechanism which (we believe) are most important to achieving customer needs. (note the use by the author to use Strong, Medium, Weak relationship indicators)

Design Space rules can be captured in these relationship and correlations values

- Identify any positive or negative correlation between the realization mechanisms.

- Note that by themselves, a realization mechanism my be favored over another, but when used in conjunction with each other may not provide the best set of realizations.
- Design Space rules can be captured in these relationship and correlations values
  - (See the DBMS vs. ASCII Text File realization comparisons)

- Implementation Difficulty of each realization is also recorded in the matrix.
  - Design Space rules can be captured in these values
- Technical Importance rating of each customer need is also recorded in the matrix.
  - Design Space rules can be captured in these values
- Matrix Analyzed and the "best" realizations are selected for the next level of decomposition



Customer Requirements	Importance	Arch Style Variant (1)	Arch Style Variant (n)
Implementation Difficulty (not considered in calc)		10	2
Req A	5	H (6) (30)	M (4) (20)
Req B	6	L (3) (18)	H (6) (36)
:			
Objective Target Values			
Absolute Technical Importance		48	56
Relative Technical Importance		2	1

- Quantified Design Space
  - A way of presenting design trade study and the rationale for selecting a design.

- Organizes the choices available for a particular design into a hierarchy of dimensions and realization alternatives.
- First correlate Requirements, to Realizations to Functional Dimensions

Concepts	Priority	Functional Dimension #1 (HW-SW-People)			Functional Dimension # (n)		
		Realization Alt #1	Realization Alt # (:)	Realization Alt #(n)	Realization Alt #1	Realization Alt # (:)	Realization Alt #(n)
Req. #1	10	1	9	2			
	:	:	:	:			
Req. #(n)	3	3	1	4			
Total Weight		19	93	20			
Req #1		10	90	20			
Req. #(:)		:	:	:			
Req #(n)		9	3	12			
Realization Design Decision		-	1	-			

- Then correlate functional dimension alternatives to structural dimension alternatives:

We could also break out the results by Requirement (which best combination of functional and structural dimensions best suits a given requirement)

– Conclusion

- Additional correlation functions can be developed and then use a statistical analysis method for selecting the best alternatives.
  - Spectrum Analysis:
    – Measures the overall goodness of a set of design decisions.
    – Note the use of both CONFORMING and DISCONFORMING measures.
  - Contribution Analysis:
    – Identifies the degree to which a system can be improved if the design choice in the ith dimension is changes to the best choice.. Therefore the dimensions with the largest

contribution indices are the best candidates for improvement.

- Design Selection Analysis:
  - Computes the number of dimensions where a particular system implements the best design choices.
- Direct Comparison Analysis:

Compares two sets of design choices to determine the amount by which one is an improvement over another.

QFD PROCESS

Quality function deployment (QFD) is a method developed in Japan beginning in 1966 to help transform the voice of the customer [VOC] into engineering characteristics for a product.[1][2] Yoji Akao, the original developer, described QFD as a "method to transform qualitative user demands into quantitative parameters, to deploy the functions forming quality, and to deploy methods for achieving the design quality into subsystems and component parts, and ultimately to specific elements of the manufacturing process." The author combined his work in quality assurance and quality control points with function deployment used in value engineering.

House of quality

A house of quality for enterprise product development processes

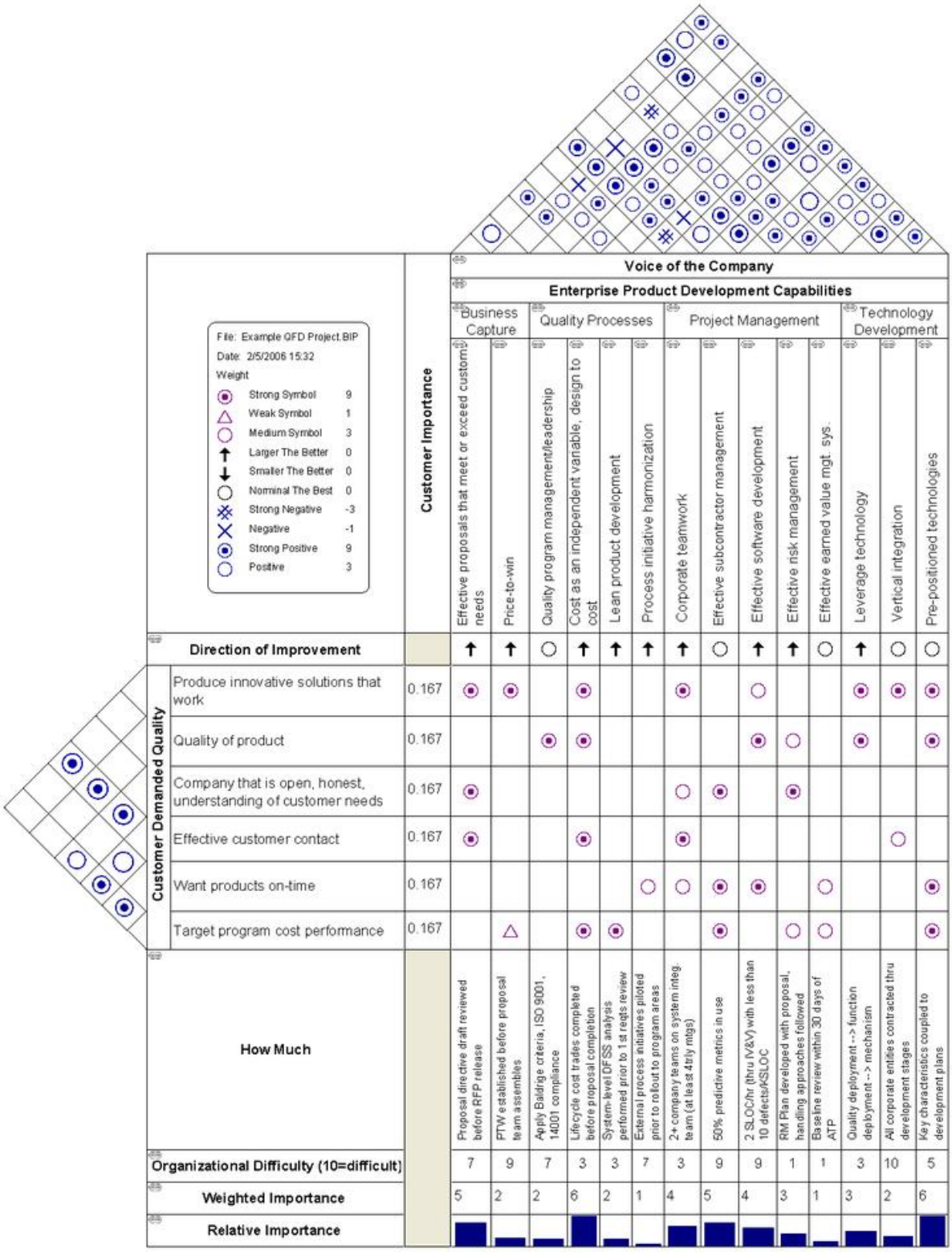
The house of quality, a part of QFD, identifies and classifies customer desires, identifies the importance of those desires, identifies engineering characteristics which may be relevant to those desires, correlates the two, allows for verification of those correlations, and then assigns objectives and priorities for the system requirements. This process can be applied at any system composition level (e.g. system, subsystem, or component) in the design of a product, and can allow for assessment of different abstractions of a system. The house of quality appeared in 1972 in the design of an oil tanker by Mitsubishi Heavy Industries.

The output of the house of quality is generally a matrix with customer desires on one dimension and correlated nonfunctional requirements on the other dimension. The cells of matrix table are filled with the weights assigned to the stakeholder characteristics where those characteristics are affected by the system parameters across the top of the matrix. At the bottom of the matrix, the column is summed, which allows for the system characteristics to be weighted according to the stakeholder characteristics System parameters not correlated to stakeholder characteristics may be unnecessary to the system design and are identified by empty matrix columns, while stakeholder characteristics (identified by empty rows) not correlated to system parameters

indicate "characteristics not address by the design parameters". System parameters and stakeholder characteristics with weak correlations potentially indicate missing information, while matrices with "too many correlations" indicate that the stakeholder needs may need to be refined.

Areas of application

QFD is applied in a wide variety of services, consumer products, and military needs.


File: Example QFD Project.BIP
Date: 2/5/2006 15:32
Weight
Strong Symbol 9
Weak Symbol 1
Medium Symbol 3
Larger The Better 0
Smaller The Better 0
Nominal The Best 0
Strong Negative -3
Negative -1
Strong Positive 9
Positive 3
Customer Importance
Voice of the Company
Enterprise Product Development Capabilities
Business Capture
Quality Processes
Project Management
Technology Development
Effective proposals that meet or exceed customer needs
Price-to-win
Quality program management/leadership
Cost as an independent variable, design to cost
Lean product development
Process initiative harmonization
Corporate teamwork
Effective subcontractor management
Effective software development
Effective risk management
Effective earned value mgt. sys.
Leverage technology
Vertical integration
Pre-positioned technologies
Direction of Improvement
Customer Demanded Quality
Produce innovative solutions that work 0.167
Quality of product 0.167
Company that is open, honest, understanding of customer needs 0.167
Effective customer contact 0.167
Want products on-time 0.167
Target program cost performance 0.167
How Much
Proposal directive draft reviewed before RFP release
FTW established before proposal team assembles
Apply Baldrige criteria, ISO 9001, 14001 compliance
Lifecycle cost trades completed before proposal completion
System-level DFSS analysis performed prior to 1st reqts review
External process initiatives piloted prior to rollout to program areas
2+ company teams on system integ. team (at least 4/rly mtgs)
50% predictive metrics in use
2 SLOC/hr (thru IV&V) with less than 10 defects/KSLOC
RM Plan developed with proposal, handling approaches followed
Baseline review within 30 days of ATP
Quality deployment --> function deployment --> mechanism
All corporate entities contracted thru development stages
Key characteristics coupled to development plans
Organizational Difficulty (10=difficult)
7 9 7 3 3 7 3 9 9 1 1 3 10 5
Weighted Importance
5 2 2 6 2 1 4 5 4 3 1 3 2 6
Relative Importance

THE VALUE OF ARCHITECTURAL FORMALISM

Formalisms

- Formal models and techniques are cornerstones of a mature engineering discipline
- Engineering disciplines used models and techniques in different ways
  - Provide precise, abstract models
  - Provide analytical techniques based on models
  - Provide design notations
  - Provide basis for simulations …

Architecture of a specific system

- Allow the architect to plan a specific system
- Becomes part of the specification of the system
  - Augments the informal characteristics of the SA
  - Permits specific analyses of the system

Architectural style

- Describe architectural abstractions for families of systems
- Purposes:
  - Make common idioms, patterns and reference architectures precise
  - Show precisely how different architectural representations can be treated as specializations of some common abstraction

Theory of software architecture

- Clarify the meaning of generic architectural concepts
  - Architectural connection, hierarchical architectural representation, architectural style
- Provide deductive basis for analyzing systems at an architectural level
  - Might provide rules for determining when an architectural description is well formed
  - Compositionality

Formal semantics of ADL:s

- Architectural description is a language issue
- Apply traditional techniques for representing semantics of languages

FORMALISMS IN USE FOR ARCHITECTURE

Formal foundation for software architecture?

- Architectural paradigms are often understood in an idiomatic way
- And applied in an ad hoc fashion

Formalisms

- Formal models and techniques are cornerstones of a mature engineering discipline
- Engineering disciplines used models and techniques in different ways
  - Provide precise, abstract models
  - Provide analytical techniques based on models
  - Provide design notations
  - Provide basis for simulations …

Architectural formalism?

- *Architecture of a specific system*

  - Allow the architect to plan a specific system
  - Becomes part of the specification of the system
    - Augments the informal characteristics of the SA
    - Permits specific analyses of the system

Architectural style

- Describe architectural abstractions for families of systems
- Purposes:
  - Make common idioms, patterns and reference architectures precise
  - Show precisely how different architectural representations can be treated as specializations of some common abstraction

Theory of software architecture

- Clarify the meaning of generic architectural concepts
  - Architectural connection, hierarchical architectural representation, architectural style
- Provide deductive basis for analyzing systems at an architectural level
  - Might provide rules for determining when an architectural description is well formed
  - Compositionality

Formal semantics of ADL:s

- Architectural description is a language issue
- Apply traditional techniques for representing semantics of languages
- How to formalize?

- How to compare their relative benefits?
- Today three (3) examples
  - Use the specification language Z

Formal SA of specific systems

- Many software systems start informally
  - There may be no other ways for this level
  - Modularization facilities of programming languages are often inadequate
    - Require designer to translate architectural abstractions to low-level primitives of programming languages
- E.g. OO architectures appropriate for some architectural decomposition
  - Still too low level of abstraction
- Use a formal specification language to describe the architecture of a system
  - High level of abstraction
- Purpose:
  - Provide a precise characterization of the system-level functions
- The example: *Oscilloscope* (see lecture on 29.1)

Oscilloscope: SA formalization

- Graph of transformations
  - Pipe-and-filter style
  - Analog signals enter the system
  - Pass through a network of transformations

- Emerge as pictures and measurements displayed front panel of instrument

- More
  - Each transformer has interface
    - User can tune transformation by configuring parameters

Oscilloscope: what to specify

- Each of the component transformations
  - Filters
- How they are interconnected
- What data is communicated between them

Data streams of oscilloscope

- To formalize the functions of the oscilloscope, begin by characterizing the data
  - Signals S, waveforms W, traces T

as functions over time, volts, and screen coordinates

- In Z
  - *Signal == AbsTime --> Volts*
  - *Waveform == AbsTime -|-> Volts*
  - *Trace == Horiz -|-> Vert*

Functions of oscilloscope

- Provide a formal description to each component
  - Explains the configuration parameters
  - What function is computed by the transformation
    - For each configuration

- Treat each component as a higher-order function
    - When applied to its configuration parameters it produces a new function representing the results of the transformation

  Oscilloscope function: Couple

- The component *Couple*
    - Used to subtract a DC offset from a signal
    - User has three (3) parameters to choose from
        - DC, AC, GND (Ground)
    - DC leaves the signal unchanged
    - AC subtracts the appropriate DC offset
    - GND produces a signal whose value is 0 volts at all times

  Oscilloscope function: Couple 2

- *Coupling* will be the type of the first parameter of the higher-order function *Couple*

  Coupling ::= DC | AC | GND

  determines the resulting function, of type

  Signal --> Signal:

  Couple: Coupling --> Signal --> Signal

  Oscilloscope function: Acquire

- A *Waveform W* is obtained from a *Signal S* by extracting a time slice
    - Waveform identical to the signal except that defined only over a bounded interval
    - Interval determined by
        - Two time values, *delay* and *duration*

- A reference time, *trigger event*

- Duration determines length of interval

- Delay determines when the interval is sampled relative to trigger event

TriggerEvent == RelTime

Oscilloscope: connectors

- Putting things together by interpreting connectors of the architecture as establishing input/output relationships between components

  - Collect the individual components and compose them into a single subsystem

  - Package the parameters of the components

  - Subsystem is a functional composition of the individual transformers

Oscilloscope: putting things together

- Package individual components parameters as single *data structure*

- *ChannelParameters*

  c:Coupling

  delay,duration:RelTime

  scaleH:RelTime

  scaleV:Volts

  posnV:vert

  posnH:Horiz

Oscilloscope: putting things together

- Subsystem

  ChannelConfiguration : ChannelParameters --> TriggerEvent -->Signal --
  >Trace

ChannelConfiguration = (λ trig:TriggerEvent ·

Clip ◦

WaveformToTrace(p.scaleH,p.scaleV,p.posnH,p

.posnV) ◦

Acquire(p.delay,p.duration) trig ◦ Couple p.c)

Oscilloscope results

- What good is this?
  - We have a precise characterization of the system
  - The architecture has been exposed as a configuration of components (parameterized data transformers) connected functionally by inputs and outputs
    - Without translating into some specific programming language
    - Makes precise certain architectural assumptions
      - Components share data only via their connections
      - External parameters need to be evaluated before the components can perform their primary function

FORMALIZING AN ARCHITECTURAL STYLE

- Problems with the previous specification:
  - The underlying architectural style is not made explicit
    - Must be inferred from the description of a particular system
      - How to elaborate the design?
      - E.g.: absence of cycles: in this example or essential feature?
  - Avoids design issues due to high abstraction level
    - How is data transmitted?

- Is there a fair scheduling between filters?

- Architectural connection is implicit via functional composition
  - Cannot reason about topological properties independently

Formalizing *pipe-and-filter style:*

components

- Components:
- *Filters* which transform streams of data
- Each filter has *input ports* for reading data and *output ports* for writing results
- A filter performs its computations incrementally and locally
- Filters operate concurrently

Formalizing *pipe-and-filter style:*

connectors

- Connectors
- *Pipes* that control the flow of data
- Each pipe links an output port to an input port
- Indicates how data flows
- Carries out the transmission

Formalizing *pipe-and -filter style:* computational step

- **A computational step is either:**
  - **An incremental transformation of data by a filter or**
  - **A communication of data between ports by a pipe**

Formalizing *pipe-and-filter style*

- A tree step approach
  - Define components: filters
  - Define connectors: pipes
  - Show how pipes and filters are combined
- For each aspect characterize its
  - Static and dynamic properties
  - Here via system state

Formalizing *pipe-and-filter style:*

data

- Given

  FILTER, PORT, FSTATE, DATA

  Port_State == PORT -|-> seq DATA

  Partial_Port_State == PORT -|-> seq DATA

Formalizing *pipe-and-filter style*

filter

- Formal filter
  - Defined by name, ports and program
  - Ports defined as a set of names
    - Directional: input ports and output ports
    - Typed ports
      - The type represents the kind of data the filter is prepared to process on that port
      - The types are subsets of DATA, the alphabet of the port

Formalizing *pipe-and-filter style*

filter 2

- Filter's program in three parts
  - Set of legal program states
  - A starting state
  - A mapping from inputs to output
    - With a possible state change as a side effect
- Gives a state machine view of a filter
- Invariant includes consistency checks:
  - Respecting port types
  - No illegal states

Formalizing *pipe-and-filter style*

filter 3

- State of the filter is composed of
  - the current program state: internal_state
  - the data in the input ports not yet read
  - data written on output ports not yet delivered

Formalizing *pipe-and-filter style*

filter 4

- A computational step of a filter
  - Reading from the inputs and writing to the outputs
  - Relation based on inputs, internal state and program

- Filter operates incrementally and locally
  - Output depends on what is actually consumed, not on data yet to be consumed
  - Is allowed to depend on historical data, not on anything outside the filter or on previous output

Formalizing *pipe-and-filter style*

pipe

- Formal pipe
  - typed connection between two ports, one output of a filter and the other an input to a filter
- State divided into two parts:
  - Data already delivered to the sink
  - Data yet to be delivered
- Pipes are here self-contained entities and
  - One can reason about them independently of filters

Formalizing *pipe-and-filter style*

pipe 2

- A consequence:
  - The same data appears in two places, at the ports of the filters and at the ends of the pipes
  - Not a problem in mathematics
  - Need to be combined when building a system

Formalizing *pipe-and-filter style*

pipe 3

- A computational step
    - Pipe delivers data from its source to its sink
- Several aspects of pipes made formal:
    - Data is not altered during transmission
    - The order of transmitted data is not changed
    - A pipe connects exactly two ports
    - The amount of data is not specified
        - Allows several different implementations and data transmission policies

Formalizing *pipe-and-filter style*

pipe-and-filter system

- Pipe-and-filter system composed as a collection of filters and a collection of pipes
- Consistency guaranteed by
    - Each filter has a unique name
    - No "dangling" pipes
        - Requirement for defining a system w/o reference to other systems
        - Pipes create a context within which filters operate
            - Pipe defines and is defined by the ports it connects
    - Ports connect to no more than one pipe
        - Distinction between filters and pipes

Formalizing *pipe-and-filter style*

pipe-and-filter system 2

- Not every port of a filter must be connected to some pipe

- Allows us to model systems that are later connected to other systems
- Open pipe-and-filter systems
- Allows hierarchical treatment of pipe-and-filter systems
  - Any pipe-and-filter system is equivalent to a high-level filter

38

Formalizing *pipe-and-filter style*

pipe-and-filter system 3

- The state of a system defined by states of its components
  - We identify the states of ports and pipes

Formalizing *pipe-and-filter style*

pipe-and-filter system 4

- A computational step is either
  - A computation of a filter or
  - A transmission of a pipe
- Non-deterministic execution of a single filter, leaving the rest of the system unchanged
- Non-deterministically chosen transmitting pipe leaving everything else unchanged

Formalizing *pipe-and-filter style*

pipe-and-filter system 5

- System computation is a sequence of steps beginning with a start state and continuing via legal computation steps
  - Every filter is in its start state
  - Every pipe is empty
  - Every output port contains no data

- Unconnected ports are not required to be empty as treated as system input/output

Formalizing *pipe-and-filter style*

pipe-and-filter system final

- Provides a precise, mathematical description of a family of systems
  - Expose essential characteristics, hiding unnecessary details
- Allows us to analyze properties of systems designed in this style
  - E.g. subnets can be encapsulated as new filters
- Specializations of the style possible
  - A pipeline

FORMALIZING AN ARCHITECTURAL DESIGN SPACE

- Problem: different designers may interpret an architectural idiom in different ways
  - Client-server might mean different things to different designers
- Related problem: several systems may be designed with similar architectural structure, but designers do not recognize this
  - Missed opportunities to share experience

- An architectural formalism can make relationships between architectures precise
- Example:
  - Relate systems built around the implicit invocation architectural style
  - Implicit invocation: components announce events
  - Components register to receive events

- When an event is announced, all the procedures associated with it will be automatically invoked

Implicit invocation

- What is the vocabulary of events?
- Can events announcements carry data?
- Concurrency in handling events?
- …
- Different answers lead to architectures with different properties

Implicit invocation 2

- How to formalize the above?
  - Start with a simple architectural abstraction and show how specific systems refine the abstraction
- Assume a basic set of events, methods, and component names

  [EVENT, METHOD, CNAME]
- An architectural component is an entity that has a name and an interface consisting of a set of methods and a set of events

Implicit invocation 3

- An event or a method has a component name and the event or method itself

  Events == CNAME × EVENT Methods
  == CNAME × METHOD
- An event system EventSystem consists of a set of components and an event manager EM
- EM associates events with methods
- The invariant asserts that

- Components have unique names
- Event manager relates actual events to actual methods

- EM is very general
  - Allows one event to be associated with many different methods
  - Even within the same component
  - Some events might be associated with no methods
  - Open issues
    - Which components can announce events?
    - Any restrictions on methods that can be associated to events?
  - The issues above need to be resolved for a more concrete architectural style

Z NOTATION

The formal specification notation Z (pronounced "zed"), useful for describing computer-based systems, is based on Zermelo-Fraenkel set theory and first order predicate logic.

•It has been developed by the Programming Research Group (PRG) at the Oxford University Computing Laboratory (OUCL) and elsewhere since the late 1970s, inspired by Jean-Raymond Abrial's seminal work.

•Formalizing an Architecture Style

• Problems with "Z"

•(1) The underlying architectural style is NOT expressed explicitly and must be inferred and therefore opens up questions in the realization of a design.

•(2) The high level abstraction avoids many design issues:

•What are the Functional, Data, and Control Architectures (topologies) !!!

•(3) Architectural Connection is Implicit:

• How are the components to be connected **?**

• How do we reason about the above topologies (architectures) independent of the system specification ?

Therefore, we need to EXPLICITLY formalize an Architecture Style.

•Notice the combination of "Z" and a MIL to define an Architecture Style in your text.

• Roz:

– The Production of formal Z specifications from annotated UML diagrams

•"RoZ automatically generates the Z schemas skeletons corresponding to a UML class diagram. To have a complete Z specification, you must add information like the definition of the type of the attributes and the constraints on the class diagram.

In the Rational Rose tool, each concept is complemented by a specification form to express additional information. We use these forms to express constraints, pre and post-conditions of operations etc. Constraints are written in the Z latex syntax in (the) "Documentation" fields of forms."

- Is used to test the results
- Independent of program code
- Mathematical Data model
- Represent both static and dynamic aspects of a system
- Decompose specification into small pieces (Schemas)
- Schemas are used to describe both static and dynamic aspects of a system
- Data Refinement
- Direct Refinement
- You can ignore details in order to focus on the aspects of the problem you are interested in Schema

Static Aspect

- The state can occupy.
- The invariant relationships that are maintained as the system moves from state to state.

POSSIBLE QUESTIONS

PART – B

1. Elucidate the Design Spaces and Rules with examples.
2. Explain the Design Rules for User Interface Architecture
3. Discuss the Quantified Design Space with examples.
4. Elaborate about Formalizing an Architectural Style with neat sketch.
5. Elucidate in detail about Formalizing an Architectural Design Space.
6. Discuss in detail about Filters in Formalizing an Architectural Style with neat sketch.
7. Write in detail on Pipes and Filters in Formalizing an Architectural Style?
8. Discuss in detail about the Value of Architectural Formalism.

PART – C

1. Explain about the Quantified Design Space with examples.
2. Elucidate in detail about Formalizing an Architectural Design Space.
3. Describe the Design Spaces and Rules with examples

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS: III BSC CS SUBJECT NAME: INTRODUCTION TO SOFTWARE ARCHITECTURE
SUBJECT CODE: 17CSP204 BATCH-2017-2019

UNIT: IV (ONE MARKS) PART A - ONLINE EXAMINATION

S.NO	QUESTION	CHOICE 1	CHOICE 2	CHOICE 3	CHOICE 4	ANSWER
1	The complete design space contains _____ functional dimensions	6	25	10	20	25
2	_____ consists of code specific to one particular program, not intended to reused in other applications	Application Specific	Shared user Interface	device dependent	None of these	Application Specific
3	_____ consists of code that specifies particular I/O devices	Shared user Interface	device dependent	Application Specific	Application Interface	device dependent
4	_____ group covers development cost consideration primarily, the required degree of adaptability of the system	External requirement	Basic interactive behavior	practical consideration	None of these	practical consideration
5	_____ mechanisms are popular for dealing with incrementally updated displays.	Standard process	Event handlers	state based	None of these	state based
6	In _____ there is no separation between application specific and shared code.	Monolithic Program	Tool Kit	Interaction Manager	Abstract device	Monolithic Program
7	A class of devices covered with screen size, number of colors, are called_____	Parameterized device	Ideal device	Variable Operations	Ad hoc devices	Parameterized device
8	_____ is the key dimension describing the device interface.	Standard process	Tool kit	Abstract device variability	Device Specific	Abstract device variability
9	The major changes in surface user interface behavior across devices is _____	Global behavior changes	Application Semantics	Uni Processing	None of these	Global behavior changes
10	A non procedural specification separate from the body of application program is called_____.	External declarative notation	External Procedural notation	Internal declarative notation	None of these	External declarative notation

11	The _____ is useful in its own right as a shared vocabulary for describing and understanding systems	design space	Rules	Design	Designer	design space
12	The underlying goal of _____ work is to organize and express software design knowledge in a useful form	creating a system design	properties of a design	reduce the effort	Utility of Codified Knowledge	Utility of Codified Knowledge
13	Programmers learned to perceive control flow in terms of a few standard concepts.	conditionals	iterations	subroutine calls	All the above	All the above
14	Software engineering matures and research for codification occurring for larger _____.	Software entities	Problems	Modules	subproblems	Software entities
15	Each dimension of a design space describes variation in one _____	system characteristic	design space	Rules	Design	system characteristic
16	The dimensions that describe functional requirements is _____	functional design space	structural design space	distance measures	performance	functional design space
17	The dimensions that describe structural choice requirements is _____	functional design space	structural design space	distance measures	performance	structural design space
18	UIMS means _____	User Interface Management System	User Information Management System	User Interconnect Management System	User Inter Management System	User Interface Management System
19	_____ consists of code that is specific to a particular I/O device class.	Device Interface	Application Interface	device-dependent component	Interface	device-dependent component
20	_____ is an example of a dimension reflecting an application-imposed external requirement	External event handling	customizability	adaptability	Interface	External event handling
21	_____ is the shared code that controls both local and global interaction sequences and stylistic decisions	components	modules	Interaction manager with fixed data types	Interaction manager with extensible data types	Interaction manager with fixed data types

22	_____ is a communication between the application and shared code is in terms of abstract information transfers.	Interaction manager with fixed data types	Interaction manager with extensible data types	Transferable abstraction	Extensible interaction manager	Extensible interaction manager
23	Aesop is a _____ language	programming language	Architecture Description Language	Software Language	Requirement language	Architecture Description Language
24	Languages, models, and formalisms can be _____ in a number of different ways	evaluated	defined	designed	described	evaluated
25	ADL Means	Architecture Description Language	Architecture Describing Language	Architecture Design Language	Architecture Designed Language	Architecture Description Language
26	_____ are being developed to make software designers more effective	Requirement language	Modeling language	Architecture Description Language	Software Language	Architecture Description Language
27	_____ are complementary to structural or framework models	Abstraction	Structures	Analysis capabilities	Dynamic models	Dynamic models
28	_____ is a minority regards architecture as a set of functional components, organized in layers that provide services upward	Functional Models	modeling distributed systems	structural models	Dynamic models	Functional Models
29	_____ is changes in small details of behavior occur across devices	Local behavior changes	Global behavior changes	Application semantic changes	None	Local behavior changes
30	In _____ only one application executes at a time	Local behavior changes	Global behavior changes	Multiprocessing	Uniprocessing	Uniprocessing
31	In _____ there are changes in underlying semantics of commands	Local behavior changes	Global behavior changes	Application semantic changes	None	Application semantic changes
32	In _____ multiple applications execute concurrently	Local behavior changes	Global behavior changes	Multiprocessing	Uniprocessing	Multiprocessing

33	_____ class identifies the basic kind of interaction supported by the user interface system.	Basic interface	application class	user interface class	none of these	Basic interface
34	_____ is based on repeated selection from groups of alternatives, at each step the alternatives are displayed	Menu selection	Form filling	Command language	Natural language	Menu selection
35	_____ is based on entry (usually text entry) of values for a given set of variables	Menu selection	Form filling	Command language	Natural language	Form filling
36	_____ is based on an artificial, symbolic language; often allows extension through programming-language-like procedure definitions.	Menu selection	Form filling	Command language	Natural language	Command language
37	_____ is based on (a subset of) a human language such as English and Resolution of ambiguous input is a key problem.	Menu selection	Form filling	Command language	Natural language	Natural language
38	_____ based on direct graphical representation and incremental manipulation of the program's data	Direct manipulation	Form filling	Command language	Natural language	Direct manipulation
39	_____ is an example of a user-imposed external requirement.	User customizability	Local behavior changes	Global behavior changes	Multiprocessing	User customizability
40	_____ group considers the data representations used within the system	User customizability	Direct manipulation	Representation issues	Command language	Representation issues
41	_____ is the shared code is simply a device driver, presenting an abstract device for manipulation by the application	Monolithic Program	Tool Kit	Interaction Manager	Abstract device	Abstract device
42	_____ is the requirement of the particular applications, users and I/O devices	External requirements	Internal Requirements	Software Requirements	Program Requirements	External requirements
43	_____ is the Key decisions about user-interface behavior that influence internal structure	External requirements	Basic interactive behavior	practical consideration	Program Requirements	Basic interactive behavior
44	_____ is the shared code provides a library of interaction techniques (e.g., menu or scroll bar handlers).	Monolithic Program	Tool Kit	Interaction Manager	Abstract device	Tool Kit

45	External event handling is an example of a dimension that reflects an application imposed external requirements.	External requirements	Basic interactive behavior	practical consideration	Program Requirements	External requirements
46	_____ provided operations and their results are well specified in terms of an "ideal" device	Ideal device	parameterized device	i/o device	Abstract device	Ideal device
47	_____ isa well-de fined set of device operations exists, but the device dependent code has considerable way in choosing how to implement the operations.	Device with variable operations	Ideal device	parameterized device	i/o device	Device with variable operations
48	Alphanumeric terminals are example of _____	Ideal device	parameterized device	Adhoc device	Abstract device	Adhoc device
49	_____ definition is a representation dimension	Representation Issues	Notation for User Interface	Both A and B	None of these	Notation for User Interface
50	Implicit in shared user interface code Information is _____	Wired into shared code	Information Buried	Application Specific	methods	Wired into shared code
51	Implicit in application code Information is _____	Wired into shared code	Information Buried	Application Specific	methods	Information Buried
52	Formalisms of this kind allow the software architect to plan a particular system	An Architecture style	The architecture of a specific system	A theory of software architecture	Formal semantics	The architecture of a specific system
53	Formalisms of this kind can be used to describe architectural abstraction for families of systems.	An Architecture style	The architecture of a specific system	A theory of software architecture	Formal semantics	An Architecture style
54	Formalisms of this kind can clarify the meaning of generic architectural concepts	An Architecture style	The architecture of a specific system	A theory of software architecture	Formal semantics	A theory of software architecture

55	This kind of formalism treats architectural description as a language issues	An Architecture style	The architecture of a specific system	A theory of software architecture	Formal semantics	Formal semantics
56	_____ represents the inputs to the oscilloscope.	signals	Waveforms	traces	All the above	signals
57	A _____ is defined by its name, ports and its program.	Filters	Pipes	Patterns	Design Space	Filters
58	A _____ is simply a typed connection between two ports.	Filters	Pipes	Patterns	Design Space	Pipes
59	MVC Means	Model View Controller	Model Viewed Controller	Modeled View Controller	Modeled View Control	Model View Controller
60	The _____ is a mathematical language developed mainly by the Programming Research Group.	schema	Z notations	Logical connectives	Proceedings	Z notations

UNIT-V

Requirements for Architecture – Description Languages – First class connectors – Adding Implicit Invocation to Traditional Programming Languages – Tools for Architectural Design – UniCon – Exploiting Style in Architectural Design Environments – Beyond definition/Use: Architectural Interconnection

REQUIREMENTS FOR ARCHITECTURE

Introduction

A critical issue in the design and construction of any complex software system is its architecture: that is, its gross organization as a collection of interacting components. A good architecture can help ensure that a system will satisfy key requirements in such areas as performance, reliability, portability, scalability, and interoperability. A bad architecture can be disastrous.

Much has changed in the past decade. Although there is wide variation in the state of the practice, generally speaking, architecture is much more visible as an important and explicit design activity in software development. Job titles now routinely reflect the role of software architect; companies rely on architectural design reviews as critical staging points; and architects recognize the importance of making explicit tradeoffs within the architectural design space. In addition, the technological basis for architectural design has improved dramatically.

Three of the important advancements have been the development of architecture description languages and tools, the emergence of product line engineering and architectural standards, and the codification and dissemination of architectural design expertise.

LINGUISTIC CHARACTER OF ARCHITECTURAL DESCRIPTION

- Common patterns in different architectures
- common kinds of elements
- common inter-module connection strategies

- Languages describe complex relations among *primitive elements* and *combinations* of these Semantic constructs

=> There is an appropriate linguistic basis in architectural descriptions

Common patterns of SW organization

- SA description often
- Box-and-line diagrams
- *boxes* □ major components
- *lines* □ communication, control, data relation
- Boxes and lines may mean different things
- For different described systems
- For different people
- Supplemented with prose, no precise meaning
- Informal terms
- Still useful

Usage of common patterns

- Informal terms
- Often refer to common patterns used to organize the system
- Used among SW engineers in high-level descriptions of designs
- More precise definitions of these
- Beneficial for SW developers
- In the forms in which they appear
- In the classes of functionality and interaction they provide

Common component classes

- **(pure) Computation**
- Simple input/output relations, no retained state
- Exp: Math functions, filters, transforms
- **Memory**

- Shared collection of persistent structured data
- Exp: Database, file system, symbol table, hypertext
- **Manager**
- State and closely related operations
- Exp: Abstract data type, servers
- **Controller**
- Governs time sequences of other's events
- Exp: Scheduler, synchronizer
- **Link**
- Transmits information between entities
- Exp: Communication link, user interface

Common interactions among components

- **Procedure call**
- Single thread of control passes among definitions
- Exp: Ordinary procedure call, remote procedure call
- **Dataflow**
- Independent processes interact through streams of data
- Exp: Unix pipes
- **Implicit invocation**
- Computation is invoked by the occurrence of an event; no explicit interactions among processes
- Exp: Event systems, automatic garbage collection
- **Message passing**
- Independent processes interact by explicit, discrete hand-off of data; may be synchronous or asynchronous
- Exp: TCP/IP
- **Shared data**
- Components operate concurrently (probably with provisions for atomicity) on the same data space
- Exp: Blackboard systems, multiuser databases

- **Instantiation**
- Instantiator uses capabilities of instantiated definition by providing space for state required by instance
- Exp: Blackboard systems, multiuser databases

Critical elements of a design language

- A (programming) language requires
- **Components**
- Primitive semantic elements and their values
- Exp: *integers, floating-point numbers, strings, records, arrays*
- **Operators**
- Functions that combine components
- Exp: *iteration, conditional constructs, +,-,*,/*
- **Abstraction**
- Rules for naming expressions of components and operators
- Exp: *definition of macros and procedures*
- **Closure**
- Rules to determine which abstractions can be added to the classes of primitive components and operators
- Exp: *procedures or user-defined types - first class entities*
- **Specification**
- Association of semantics to the syntactic form
- *Formal, informal (in reference manual)*

The language problem for SA

- SA deals with
- Allocation of functionality to components
- Data and communication connectivity
- Overall performance, quality attributes and system balance

- Quite different from the (conventional) programming language concerns

Critical elements of a design language for SA

Components

- Module-level elements; component classes listed before
- **Operators**
- Interaction mechanisms as listed before
- **Abstraction**
- Compositions in which code elements are connected in a particular way; Exp: *client-server relation*
- **Closure**
- Conditions in which composition can serve as a subsystem in development of larger systems
- **Specification**
- Not only of functionality, but also of quality attributes

Implication of the critical elements

- Basis for designing ADLs provided by
- Identification of architectural components
- Identification of architectural techniques, for combining them into *subsystems* and *systems*
- Such a language would support
- Simple expressions of connections among simple modules, plus
- Subsystems
- Configurations of subsystems into systems
- Common paradigms for such combinations
- Expression of quality attributes and functional properties

Requirements for ADLs

1. To provide models, notations, tools to describe architectural components and their interactions

2. To handle large-scale, high-level designs

3. To support the adaptation of designs to specific implementations

4. To support user-defined abstractions

5. To support application-specific abstractions

6. To support the principled selection of architectural paradigms

ADL and environment

- Close relation between ADL and its environment
- ADL: precise descriptions
- Environment: (re)uses the descriptions
- Ideal ADL should support
- **Composition**
- **Abstraction**
- **Reusability**
- **Configuration**
- **Heterogeneity**
- **Analysis**

Composition

- Describe a system as composition of independent components and connections
- Aspects
- Divide a complex system (hierarchically) into smaller parts

- Assemble a large system from constituent elements
- Independent elements
- Can be understood in isolation from the system
- Separate issues of implementation-level from those of architectural level

Abstraction

- Allows to describe the abstract roles of elements and their interaction within SA at a level well understood by designers
- Clearly
- Explicitly
- Intuition
- Suppress unneeded detail but reveal important properties
- Distinct roles of each element in the high-level structure are clear
- Example: client-server relationship

Reusability

- Reuse components, connectors, architectural styles in different architectural descriptions
- Reuse generic patterns of components and connectors
- Families of SA as open-ended sets of architectural elements
- Structural and semantic constraints
- Differs with respect to reusing components from libraries
- Those are completely closed / parameterized components, retain identities, are leaves of "is-composed-of" system structure
- Reusing generic patterns of components and connectors: further instantiation, indefinite replication, structured collections of internal nodes

Configuration

- Architectural descriptions should localize the description of system structure
- Dynamic reconfiguration permissible
- Evolvability
- Create/remove components, interactions
- Permit to understand and change architectural structure
- Without examining individual components
- Reason about composition as a whole
- Separate descriptions of compositions from those of elements

Heterogeneity

- Combine multiple, heterogeneous architectural descriptions
- Ability to combine different architectural styles in a single system
- Ability to combine components written in different languages

Analysis

- Possible to perform rich and varied analyses of architectural descriptions
- Each style facilitates a certain type of properties
- Automated and non-automated reasoning about architectural descriptions
- Important for architectural formalisms
- Variety of analyses => no single semantic framework will be enough
- Should be possible to associate specifications with architectures, relevant to particular components, connectors, styles

FIRST-CLASS CONNECTORS

- SA treats SW systems as composition of components
- Focus on components

- Description of interactions among components is implicit, distributed, hard to identify

=> Info organized around components, significance of interactions, connections is ignored

Problems with this practice

1. Inability to localize info about interactions
2. Poor abstractions
3. Lack of structure on interface definitions
4. Mixed concerns in programming language specification
5. Poor support for components with incompatible packaging
6. Poor support for multi-language or multi-paradigm systems
7. Poor support for legacy systems

Fresh view of software system composition

- Systems composed of identifiable **components** of various distinct types
- These interact in identifiable, distinct ways
- Correspond to compilation units (roughly)
- **Connectors** mediate interactions among components
- Establish rules that govern component interaction
- Specify any auxiliary mechanisms required
- Do not correspond to compilation units

Connectors

- Manifest as
- Table entries
- Instructions to a linker
- Dynamic data structures
- System calls
- Initialization parameters

- Servers with multiple independent connections
- Define a set of roles that specific named entities of the components must play
- Place of relations among components
- Mediate interactions
- Have protocol specifications defining their properties
- Rules about types of interfaces they are able to mediate for
- Assurances about properties of interactions
- Rules about order in which things happen
- Commitments about interaction (ordering, performance, etc)
- Are of some type/subtype
- Roles to be satisfied: specific, visible named entities in the protocol of a connector

Components

- Place of computation and state
- Have interfaces specifying their properties
- Signatures
- Functionality of resources
- Global relations
- Performance properties
- Are of some type/subtype
- Interface points: specific, visible named entities in the interface of a component

Primitive vs composite: components

- Primitive components coded in the programming language
- Composite components define configurations in independent notation
- Constituent components and connectors identified
- Match connection points of components with roles of connectors
- Check integrity of the above

Primitive vs composite: connectors

- Of different kinds
- Shared data representations
- Remote procedure calls
- Dataflow
- Document-exchange standards
- Standardized network protocols
- Rich enough set to require taxonomy to show relations among similar connector kinds

Primitive connectors

- Built-in mechanisms of programming languages
- System functions of the OS
- Shared data
- Entries in task/routing tables
- Interchange formats for static data
- Initialization parameters etc

ADDING IMPLICIT INVOCATION TO TRADITIONAL PROGRAMMING LANGUAGES

Implicit invocation based on event broadcast is an increasingly important technique for integrating systems. However, the use of this technique has largely been confined to tool integration systems - in which tools exist as independent processes – and special purpose languages – in which specialized forms of event broadcast are designed into the language from the start.

Adding Implicit Invocation to Ada

While there are many ways to implement an implicit invocation mechanism, all are based on two fundamental concepts. The first is that in addition to defining procedures that may be invoked in the usual way, a module is permitted to announce events. The second is that a module may register to receive announced events. This is done by associating a procedure of that module with each event of interest. When one of those events is announced the implicit invocation mechanism is responsible for calling the procedures that have been registered with the event.

Overview of the Implemental ion

In Ada the basic unit of modularization is the package [Ada83]. Packages have interfaces, which define (among other things) a set of exported procedures. We developed a small specification language to augment package interfaces. This language allows users to identify events they want the system to support, and to specify which Ada procedures (in which package specifications) should be invoked on announcing the event.

Key Design Questions

This simple implementation provides many characteristics of more complex implicit invocation systems. However, it embodies a set of design choices whose consequences are important to understand, both to see how to use an implicit invocation system, and to observe the limitations of the implementation. The design decisions can be grouped into the following six categories:

1. Event definition

2. Event structure

3. Event bindings

4. Event announcement 5. Concurrency 6. Delivery policy

Event Definition

The first design issue concerns how events are to be defined. There are several related issues. Is the vocabulary of events extensible? If so, are events explicitly declared? If events are declared, where are they declared? We considered three approaches to event extensibility and declaration.

Fixed Event Vocabulary A fixed set of events is built into the implicit invocation system: the user is not be allowed to declare new events.

Static Event Declaration The user can introduce new events, but this set is fixed at compile time.

Dynamic Event Declaration New events can be declared dynamically at run time, and thus there is no fixed set of events.

No Event Declarations Events are not declared at all; any component can announce arbitrary events.

Event Bindings

Event bindings determine which procedures (in which modules) will be called when an event is announced. There are two important questions to resolve. First, when are events bound to the procedures? Second, how are the parameters of the event passed to these procedures? With respect to the first issue, we considered two approaches to event binding:

Static Event Bindings Events are bound to procedures statically when a program is compiled.

Dynamic Event Bindings Event bindings can be created dynamically. Components register for events at run time when they wish to receive them, and reregister for events when they are no longer interested.

Event Announcement

Although announcing an event is a straightforward concept, there are several ways in which it can be incorporated.

Single Announcement Procedure Provide a single procedure that would announce any event. Pass it a record with a variant part containing the event type and arguments.

Multiple Announcement Procedures Provide one announcement procedure per event name. For example, to announce the Changed event a component might call Announce_ Changed. The procedure accepts exactly the same parameters (in number, type, order, and name) as the event.

Language Extension Provide an announce statement as a new kind of primitive to Ada and use a language preprocessor to conceal the actual Ada implementation.

Implicit Announcement Permit events to be announced as a side effect of calling a given procedure. For example, each time procedure Proc is invoked, announce event Event.

Concurrency

Thus far our enumeration of design decisions has left open the question of exactly what a component is. In our design, we considered three options.

Package A component is a package, and an invocation is a call on a procedure in the package interface.

Packaged Task A component is a task (with an interface in a package specification), and an invocation is a call on an entry in the task interface.

Free Task A component is a task. An invocation is a call on an entry in the task interface. However, rather than providing an enclosing package, the task is built inside the Event_Hanager package.

Delivery Policy

In most event systems, when an event is announced all procedures bound to it are invoked. However, in some event systems this is not guaranteed. While delivery policy was not a major

question in our development, there is enough variation in the way this is done in other systems to explore the design options. The ones we considered are:

Full Delivery An announced event causes invocation of all procedures bound to it.

Single Delivery An event is handled by only one of a set of event handlers. For example, this allows such events as "File Ready for Printing" to be announced, with the first free print server receiving the event. This delivery policy provides a form of "indirect invocation", as opposed to "implicit invocation".

Parameter-Based Selection This approach uses the event announcement's parameters to decide whether a specific invocation should be performed. This is similar to the pattern matching features of Field [Reiss 90] in that a single event can cause differing sets of subprograms to be invoked depending upon exactly what data is transferred with the event.

State-based Policy Some systems (notably Forest [Garlan & Ilias 91]), associate a "policy" with each event binding. Given an event of interest, the policy determines the actual effect of it.

EXPLOITING STYLE IN ARCHITECTURAL DESIGN ENVIRONMENTS

What is Architectural Style?

While there is currently no single well-accepted definition of software architecture it is generally recognized that an architectural design of a system is concerned with describing its gross decomposition into computational elements and their interactions [PW92, GS93b, GP94]. Issues relevant to this level of design include organization of a system as a composition of components; global control structures; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives.

It is possible to describe the architecture of a particular system as an arbitrary composition of idiosyncratic components. However, good designers tend to reuse a set of established

architectural organizations – or architectural styles. Architectural styles fall into two broad categories.

Idioms and patterns:

This category includes global organizational structures, such as layered systems, pipe-filter systems, client server organizations, blackboards, etc. It also includes localized patterns, such as model-view-controller [KP88] and many other object-oriented patterns [Coa92, GHJV94].

Reference models:

This category includes system organizations that prescribe specific (often parameterized) configurations of components and interactions for specific application areas. A familiar example is the standard organization of a compiler into lexer, parser, typer, optimizer, code generator [PW92]. Other reference architectures include communication reference models (such as the ISO OSI 7-l layer model.

More specifically, we observe that architectural styles typically determine four kinds of properties [AAG93]:

1. They provide a vocabulary of design elements – component and connector types such as pipes, filters, clients, servers, parsers, databases etc.

2. They define a set of configuration rules – or topological constraints – that determine the permitted compositions of those elements. For example, the rules might prohibit cycles in a particular pipe-filter style, specify that a client-server organization must be an n-to-one relationship, or define a specific compositional pattern such as a pipelined decomposition of a compiler.

3. They define a semantic interpretation, whereby compositions of design elements, suitably constrained by the configuration rules, have well-defined meanings.

4. They define analyses that can be performed on systems built in that style. Examples include schedulability analysis for a style oriented toward real-time processing [Ves94] and deadlock detection for client-server message passing [JC94].

Use of standardized styles supports interoperability. Examples include CORBA object-oriented architecture [Cor91], the OSI protocol stack [McC91], and event-based tool integration [Ger89].

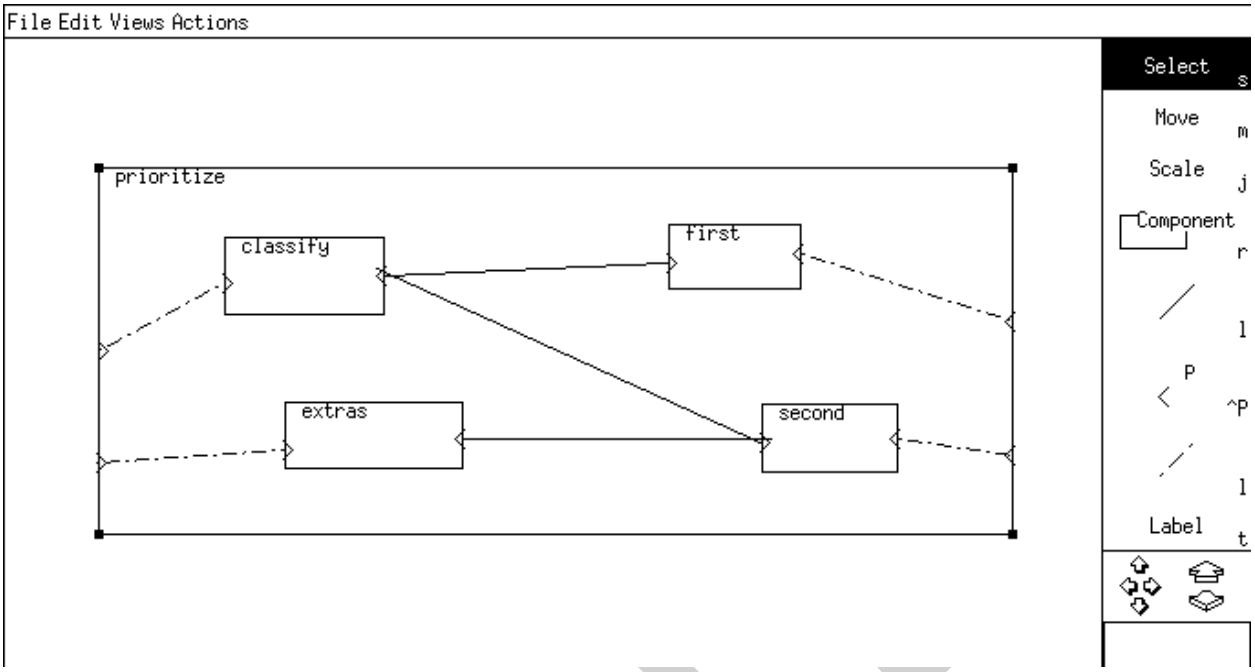
Automated Support for Architectural Design

Given these benefits, it is perhaps not surprising that there has been a proliferation of architectural styles. In many cases styles are simply used as informal conventions. In other cases – often with more mature styles – tools and environments have been produced to ease the developer's task in conforming to a style and in getting the benefits of improved analysis and code reuse.

Figure 1: Generating Fables with Aesop

Aesop

Aesop is a system for developing style-specific architectural development environments. Each of these environments supports (1) a palette of design element types (i.e., style-specific components and connectors) corresponding to the vocabulary of the style; (2) checks that compositions of design elements satisfy the topological constraints of the style; (3) optional semantic specifications of the elements; (4) an interface that allows external tools to analyze and manipulate architectural descriptions; and (5) multiple style specific visualizations of architectural information together with a graphical editor for manipulating them. Building on existing software development environment technology, Aesop adopts a "generative" approach. As illustrated in Figure 1, Aesop combines a description of a style (or set of styles) with a shared toolkit of common facilities to produce an environment, called a *Fable*, specialized to that style (or styles).



The Structure of a Fable

Aesop adopts a conventional structure for its environments: a Fable is organized as a collection of tools that share data through a persistent object base (Figure 5). The object base runs as a separate server process and provides typical database facilities: transactions, concurrency control, persistence, etc. In the initial prototype the

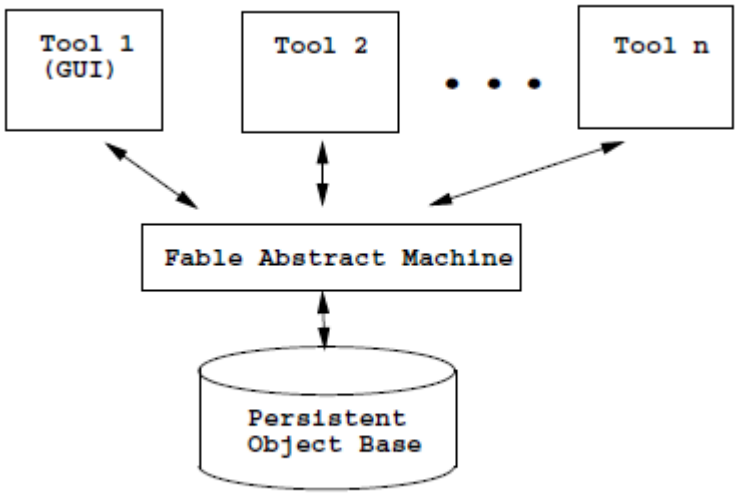


Figure 5: The Structure of a Fable

database was built by "serverizing" OBST, a public domain, C++- oriented database Tools run as separate processes and access the object base through an RPC interface called the "Fable Abstract Machine" (or FAM), which defines operations for creating and manipulating architectural objects. This interface is defined as a set of C++ object types that are linked with tools that intend to directly manipulate architectural data. Additionally, tools can register an interest in specific data objects, and will be notified when they change. Currently we use Hewlett Packard's Softbench [Ger89] for event-based tool invocation. This same mechanism also serves to integrate external tools. For example, in the pipe-filter environment, described above, code is generated by announcing a message to a suitably "encapsulated" code generation tool. Tools such as external editors are handled in the same way. The user interface to aFable is centered around a graphical editor and database browser provided by the Aesop system.

Representing Architectural Designs

Given a persistent object base for architectural representation, an important question is what are the types of objects that can be stored in the database. The answer to this question is critical, since, in effect, it answers the deeper question: what is an architectural design and how is it represented? Our approach to architectural representation is based on a generic ontology of seven

entities: components, connectors, configurations, ports, roles, representations, and bindings. The basic elements of architectural description are *components*, *connectors* and *configurations*. Components represent the loci of computation; connectors represent interactions between components; and configurations define topologies of components and connectors. Both components and configurations have interfaces. A component interface is defined by a set of *ports*, which determine the component's points of interaction with its environment. Connector interfaces are defined as a set of *roles*, which identify the participants of the interaction

Defining Styles

The generic object model provides the foundation for representing architecture. However, to obtain a useful environment, that framework must be augmented to support richer notions of architectural design. In Aesop this is done by specifying a style.

A Pipe-Filter Style

As indicated earlier, a pipe-filter style supports system organization based on asynchronous computations connected by dataflow.

Vocabulary. Figure 8 illustrates the type hierarchy we used to define a pipe-filter style. *Filter* is a subtype of component and *pipe* a subtype of connector. Further, ports are now differentiated into *input* and *output* ports, while roles are separated into *sources* and *sinks*.

Configuration rules. The pipe-filter style constrains the kinds of children and connections allowed in a system. Besides the constraints on port addition described above, pipes must take data from ports capable of writing data, and deliver it to ports capable of reading it. Hence, source roles can only attach to input ports, and sink roles can only attach to output ports. (Figure 7 shows how this constraint is enforced by a method of the new *pf source* class. Most of the configuration rules are equally simple, although some—such as prohibiting cycles—can be considerably more complex.)

Semantic interpretation. In the prototype pipe-filter the semantics of filters is given by a simple, style-specific filter language, as was illustrated in Figure 3. The associated tool (based on Gandalf [HN86]) provides typechecking and other static analyses.

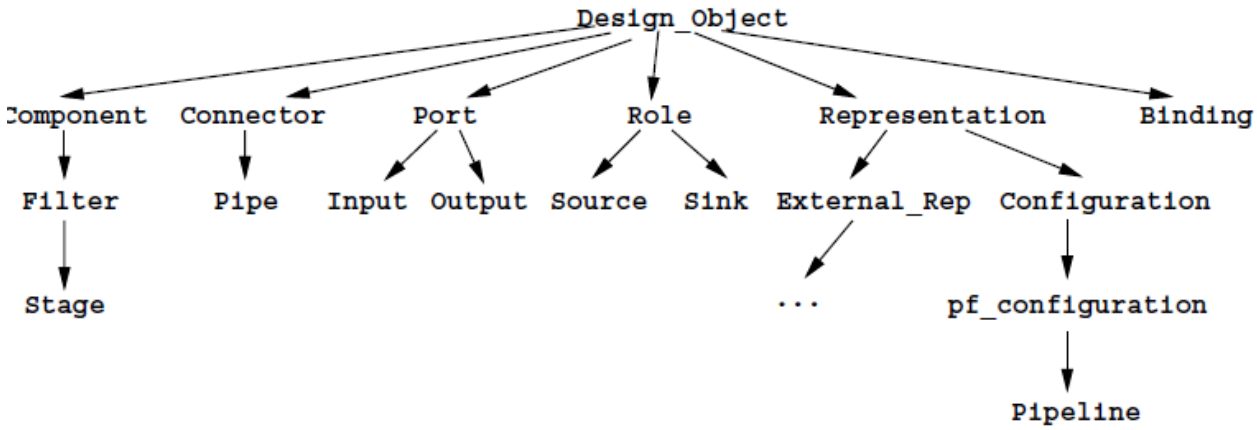


Figure 8: Style Definition as Subtyping

Analyses. In addition to the static semantic checksjust outlined, we incorporated a tool for generating code from filter descriptions. Hence, a pipe-filter description can be used to generate a running program, with the help of some style-specific tool and the external Gandalf tool.

A Pipeline Style A pipeline style is a simple specialization of the a pipe-filter style. It incorporates all aspects of the the pipe-filter style except that the filters are connected together in a linear order, with only one path of data flow. (This corresponds to simple pipelines built in the Unix shell.) The pipeline style is an example of stylistic subspecialization.

A Real-Time Style

An important class of system organization divides computations into tasks communicating by synchronous and asynchronous messages. Within this general category are systems that must satisfy real-time scheduling constraints while processing their data. We created an Aesop environment for an architectural style, developed at the University of North Carolina, that supports the design of such systems [Jef93]. Underlying the architectural style is a body of

theory for analyzing real-time systems [Jef92]. This theory allows one to determine the (scheduling) feasibility of a system from the processing rates of its component tasks, rates of inputs from external devices, and shared resource loads. The theory also leads to heuristics for improving the schedulability of a system that is not feasible.

The style has been applied primarily to real-time, multi-media applications

An Event-based Style In an event-based style, components register their interest in certain kinds of events, and then can announce events and receive them according to their interest.

Vocabulary. The event style defines a new "participant" component that registers for, announces, and receives events. An "event bus" connector is used to propagatethe events between components.

Configuration Rules. In this style, configuration rules simply state that the event bus connects only to components that announce or receive events.

Semantic Interpretation. Components are permitted to communicate events between each other only if they have a common bus to which they are connected, and the receiving component registered an interest in the type of event announced by the sending component. An announced event can be received by zero or more other components (unlike in the pipe-filter style, where written data can only be read by one other component).

Analyses. A number of analyses are possible in event-based styles, such as identifying the flow of communication between components. As in the pipe-filter style, given a language for specifying the communication behavior of participant components, a compiler can be built to generate code for a particular event-based configuration [GS93a]. (We did not do this, however, in our prototype.)

User Interface

In addition to providing a representational model for tools to create and manipulate architectural descriptions, an environment must also provide a way for the user to view, edit, and use these descriptions. As we outlined earlier, the default interface is a graphical editor,

which is automatically provided by Aesop and which runs as a separate tool in the environment. To produce a style-specific environment this editor (and potentially other interface tools) must also be specialized. To accomplish this, each architectural class is associated with one or more visualization classes. New subclasses introduced by a style inherit the visualizations of their superclass, but may also define their own visualization classes. This induces a parallel hierarchy of visualization types, in which the upper portion of that hierarchy is defined by the default visualizations for the generic architectural types.

The first category concerns the way in which styles are described, and includes:

_ Explicit representation of stylistic constraints.

Control over super type visibility.

Dynamic incorporation of style descriptions.

Type migration.

TOOLS FOR ARCHITECTURAL DESIGN

Architecture Description Languages

- The positives
- ADLs provide a formal way of representing architecture
- ADLs are intended to be both human and machine readable
- ADLs support describing a system at a higher level than previously possible
- ADLs permit analysis of architectures – completeness, consistency, ambiguity, and performance
- ADLs can support automatic generation of software systems
- The negatives
- There is no universal agreement on what ADLs should represent, particularly as regards the behavior of the architecture

- Representations currently in use are relatively difficult to parse and are not supported by commercial tools

Software Architecture: ADL Perspective

- The ADL community generally agrees that Software Architecture is a set of components and the connections among them.
- components
- connectors
- configurations
- constraints

ADLs

- **Leading candidates**
- ACME (CMU/USC)
- Rapide (Stanford)
- **Wright (CMU)**
- **Unicon (CMU)**
- **Secondary candidates**
- **Aesop (CMU)**
- MetaH (Honeywell)

Our model thus describes software systems in terms of two kinds of distinct, identifiable elements: components and connectors. Each of the two elements has a type, a specification, and an implementation. The specification defines the units of association used in system composition; the implementation can be primitive or composite. Figure 4 suggests the essential character of the model.

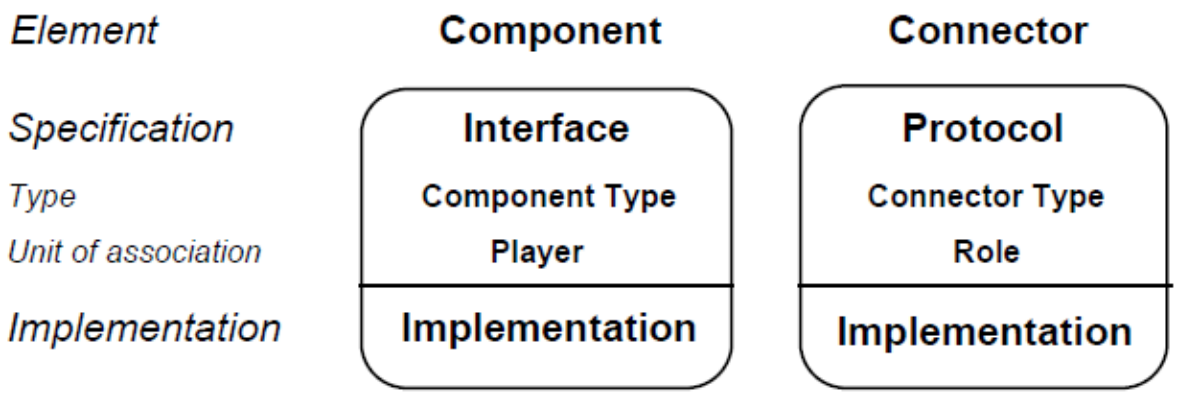


Figure 4: Gross structure of an architecture language.

Components are the locus of computation and state. Each component has an interface specification that defines its properties. These properties include the component's type or subtype (e.g. filter, process, server, data storage), functionality, guarantees about global invariants, performance characteristics, and so on. The specific named entities visible in a component's interface are its players. The interface includes the signature, functionality, and interaction properties of its players.

Connectors are the locus of definition for relations among components. They mediate interactions but are not "things" to be "hooked up;" rather, they provide the rules for hooking-up. Each connector has a protocol specification that defines its properties. These properties include its type or subtype (e.g. remote procedure call, pipeline, broadcast, shared data representation, document exchange standard, event), rules about the types of interfaces it works with, assurances about the interaction, commitments about the interaction such as ordering or performance, and so on. The specific named entities visible in a connector's protocol are roles to be satisfied. The interface includes rules about the players that can match each role, together with other interaction properties.

Components may be either primitive or composite.

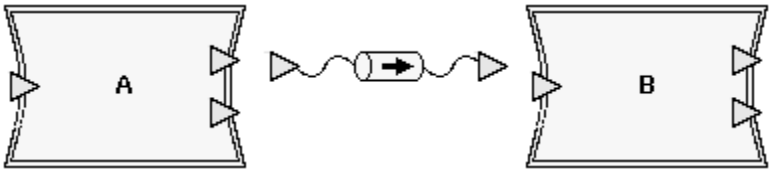
Primitive components may be implemented as code in a conventional programming language, shell scripts of the operating system, software developed in an application such as a spreadsheet, or other means external to the architectural description language.

Composite components define configurations in a notation independent of conventional programming languages. This notation must be able to identify the constituent components and connectors, match the players of components with roles of connectors, and check that the resulting compositions satisfy the specifications of both the components' interfaces and the connectors' protocols.

UniCon

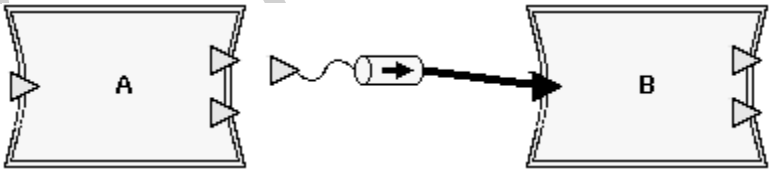
UniCon is an architectural description language whose focus is on supporting the variety of architectural parts and styles found in the real world and on constructing systems from their architecture descriptions. To give a feel for what describing an architecture is like in UniCon, here is a short example.

An architecture description in UniCon consists of a set of components and connectors. A component is a locus of data or computation, while a connector mediates the interaction among components. Each component has an interface that exports a set of players. These players engender the ways in which the component can interact with the outside world. Similarly, a connector's protocol exports a set of roles that engender the ways in which the connector can mediate interaction. To illustrate, here's an example diagram produced using UniCon's graphical editor:



The diagram features two components, labelled A and B, which are Unix filters. Each of them exports three players, drawn as triangles; the player on the left represents the input stream "standard in", while the players on the right represent the output streams "standard out" and "standard error." Between the two components is a connector, which represents a Unix pipe. The connect exports two roles: the one dangling to the left represents the pipe's source; the one dangling to the right represents the pipe's sink.

In the picture above, there is no interactive among the components and connectors; nothing is "hooked up." To specify that there should be a connection, a player must be associated with a role. In the graphical editor, this is done by dragging the role over the player and dropping it. The result of dragging the pipe's sink and dropping it on B's input is shown here:



By associating players and roles, a whole configuration of interacting parts can be specified. The current version of UniCon supports not only pipe-and-filter systems like those above, but also modules interacting with procedure calls and shared data, distributed systems with RPC calls, processes that share processors according to various real-time disciplines, and databases accessed with SQL commands.

To speed the process of gaining experience, some simplifications were made to the model when implementing UniCon, namely:

• UniCon supports composite components but not composite connectors.

• Abstractions for components and connectors are built in, but new ones cannot yet be defined. These built-in elements sample a diverse space, but the set is in no sense complete and a unifying taxonomy is not yet provided.

• The only primitive components at present are compilation units. Although the implementation is largely language-indifferent, C is the language supported at present.

• The syntax has not been refined for conciseness yet. It can be a bit wordy, especially when making intermodule connections of procedures and data with no change of name.

Abstraction and Encapsulation

For a composite element, the implementation part consists of

• a parts list (components and connectors)

• composition instructions (association between roles and players)

• abstraction mapping (relation between internal players and players of the composite)

• other related specifications (detailed properties of the parts and compositions).

Types and Type Checking

A problem similar to type checking in a programming language arises at four points in an architectural

language. Two of these appear in the preceding discussion: the types of components and of connectors and their use in showing adherence to a style. As with any type system, types for components

and connectors express the designer's intent about how to use the element properly and are most useful when the language checks them. The types for connectors and components are not merely enumerations of unrelated items; some are closely related to others. Architectural types describe

expected capabilities and limit both what can appear in the construct's specification and the legitimate ways to use the construct. Examination of real systems shows that type hierarchies of this
sort are useful. For example, there are many kinds of memories (components) and many kinds of event systems (connectors). Defining type structures for these elements requires the creation of taxonomies
to catalog and structure the type variations. This is part of establishing a full model for architectural composition.

Accommodating Analysis Tools

Architectural descriptions should be "open" with respect to analysis tools. We must accommodate
techniques that are applied at the systems level of design. These analysis tools will often be developed
independent of the model. They may address such properties as functional correctness, performance,
and timing (e.g., allowable order of operations, real-time guarantees). The architectural description language should be able to interact with any analysis technique that works with information
in the architectural specifications. It should be able to record the system-level specifications required by external tools as uninterpreted expressions, deliver information to the tools, receive results
from the tools, and incorporate those results in the architectural description.

BEYOND DEFINITION/USE: ARCHITECTURAL INTERCONNECTION

Large software systems require decompositional mechanisms in order to make them tractable. By breaking a system into pieces it becomes possible to reason about overall properties by understanding the properties of each of the parts. Traditionally, Module Interconnection

Languages (MILs) and Interface Definition Languages (IDLs) have played this role by providing notations for describing (a) computational units with well-defined interfaces, and (b) compositional mechanisms for gluing the pieces together. A key issue in design of a MIL/IDL is the nature of that glue. Currently the predominant form of composition is based on definition/use bindings [13]. In this model each module defines or provides a set of facilities that are available to other modules, and uses or requires facilities provided by other modules. The purpose of the glue is to resolve the definition/use relationships by indicating for each use of a facility where its corresponding definition is provided.

Implementation versus Interaction At the level of system design one important relationship between modules is that of "implements": a given module is defined in terms of facilities provided by other modules. For example, one module might import a string package that it uses to implement an internal data representation. But this is not the only important kind of relationship. When people design systems they typically provide an architectural description consisting of a set of computational components and set of inter-component connections that indicate the interactions between those components. Often these descriptions are expressed informally as box and line diagrams, and the interaction relationshipsare described idiomaticallywith phrases such as "client-server interaction", "pipe and filter organization", or "event-broadcast communication". In these descriptions the components are treated as independent entities that may interact with each other in complex ways [5, 11]. The distinction between a description of a system based on "implements" relationships and one based on "interacts" relationships is important for three reasons.

First, the kinds of description involve different ways of reasoning about the system

Second, the two kinds of relationship have different requirements for abstraction.

Third, they involve different requirements for compatibility checking

Example

To illustrate the distinctions outlined above consider a simple system, *Capitalize*, that transforms a stream of characters by capitalizing alternate characters while reducing the others to lowercase. Let us assume that the system is designed as a pipe-filter system that splits the input stream (using the filter *split*), manipulates each resulting substream separately (using filters *upper* and *lower*) and then remerges the substreams (using *merge*). In a typical implementation of this design we would likely find a decomposition such as the one illustrated in figure 1. It consists of a set-up routine (*main*), a configuration module (*config*), input/output libraries, as well as modules for accomplishing the desired transformations. The set-up routine depends on all of the other modules, since it must coordinate the transformations and do the necessary hooking up of the streams. Each of the filters uses the configuration module to locate its inputs and outputs, and the i/o library to read and write data.

This second description clearly highlightsthe architectural design and suggests that in order to understand a system it is important to express not only the definition/use dependency relations between implementation "modules," but also to reflect directly the abstract interactions that result in the effective composition of independent components. In particular, to understand and reason about *Capitalize* it is at least as important to know that the output of *upper* is delivered to *merge* as it is to know that it is invoked by *main* and uses *i/o library*.

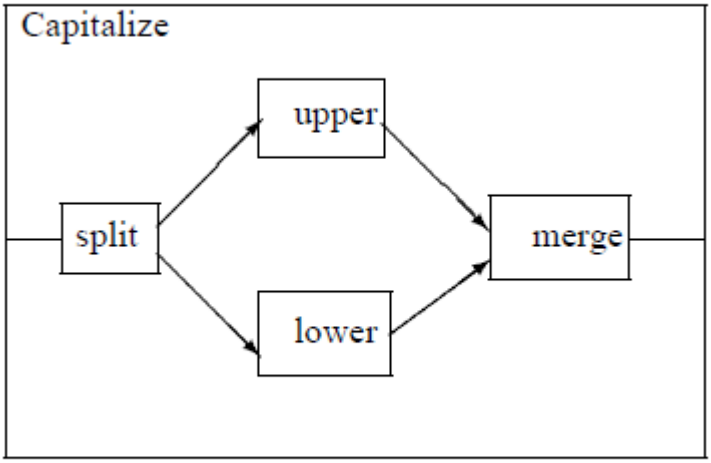


Figure 2: An Architectural Description.

TheWRIGHT **Model of Architectural Description**

Unfortunately, except in very specialized circumstances, the kind of architectural description given above is usually informal, and programmers must ultimately rely on concrete descriptions of implementation relationships to find the "truth" of the system.

In order to support more direct specification and analysis of architectural descriptions, we have developed The WRIGHT architectural specification language. WRIGHT specifications are based on the idea that interaction relationships between components of a software system should be directly specifiable as protocols that characterize the nature of the intended interaction.

WRIGHT allows one to define reusable connector classes that can be instantiated as needed to produce system descriptions. The meaning of a connector is given by a set of protocols, which state what are the roles of interaction and how these roles interact with each other.

```
System Capitalize
    Component Split
        port In [input protocol]
        port Left, Right [output protocol]
        comp spec [Split specification]
    Component Upper
        port In [input protocol]
        port Out [output protocol]
        comp spec [Upper specification]
    ...
    Connector Pipe
        [Pipe specification]
Instances
    split: Split; upper: Upper; lower: Lower; merge: Merge;
    p1,p2,p3,p4: Pipe
Attachments
    split.Left as p1.Writer;
    upper.In as p1.Reader;
    split.Right as p2.Writer;
    lower.In as p2.Reader;
    ...
end Capitalize.
```

Figure 3: *Capitalize* in WRIGHT

Figure 3 illustrates the use of the notation for describing the example system. As shown, a system description is divided into two parts. First, both component and connector classes are specified, indicating the interface and computation of components and the protocol that the connector class represents. Second, the configuration of the system is described using instances of these classes, and the architectural topology is defined as a list of attachments.

AWRIGHT specification describes a component interface as a collection of *ports*, or logical interaction points. These ports factor the expectations and promises of the component into the points of interaction through which the component will interact with its environment. The component may optionally further specify how the interactions on its ports are combined into a computation.

Reasoning About Architectural Descriptions

An important property of any system description language is its ability to support reasoning about system descriptions. Standard MILs typically support reasoning about certain forms of

consistency (such as typechecking) and (possibly) correctness. In the case of architectural connection we are interested in richer forms of consistency. Specifically, we would like to be able to know when it is legal to attach a given connector as a port in some system description. We refer to this as the "port-role compatibility" problem. (Recall that we use an instance of a connector by associating its roles with the ports of instances of components—see figure 3.)

The most obvious and constrained form of compatibility checking would be to simply check that the port and role have identical protocols. But this is too restrictive. For example, we saw above that the ports of *Split* did not take advantage of the full flexibility provided by the *Pipe* roles. Similarly, we would like to be able to connect either end of a pipe to a file (as is done in Unix), even though files support both reads and writes while the end of a pipe supports only one. As another example, it should be possible to use a server port in a role that requires fewer services than the component provides. On the other hand we would not like the client port to attempt to use more services than the server provides.

POSSIBLE QUESTIONS

PART – A

1. List out the Requirements for Architecture Description Languages.
2. Discuss in detail about First Class Connectors
3. Write in detail about UniCon: A Universal Connector Languages.
4. Discuss in detail about Wright Model of Architectural Descriptions.
5. Explain in detail Adding Implicit Invocation to Ada.
6. Explain in detail about Beyond Definition/Use: Architectural Interconnection.
7. What is Architectural Style?

PART – C

1. Explain about Requirements for Architecture Description Languages.
2. Describe about the UniCon: A Universal Connector Languages.
3. Elucidate the Wright Model of Architectural Descriptions

KARPAGAM ACADEMY OF HIGHER EDUCATION

CLASS: III BSC CS SUBJECT NAME: INTRODUCTION TO SOFTWARE ARCHITECTURE
SUBJECT CODE: 17CSP204 BATCH-2017-2019

UNIT: V (ONE MARKS) PART A - ONLINE EXAMINATION

S.NO	QUESTION	CHOICE 1	CHOICE 2	CHOICE 3	CHOICE 4
1	_____ is a system for developing style-specific architecture development environment	Aesop	Fables	Style	palette
2	_____ category includes global organizations structures	Reference Models	Idioms and Patterns	Patterns	Idioms
3	Idioms and Patterns include_____.	General Pattern	Reference Pattern	Idioms Pattern	Localized Pattern
4	_____ category includes system Organization that prescribe specific configurations of components and interactions for specific applications areas	Idioms and Patterns	General Pattern	Localized Pattern	Reference Model
5	Other Reference architecture include _____.	Communication Reference Model	Reference Model	Communication Model	User Interface Model
6	Architectural Style typically determine _____ kinds property	1	2	3	4
7	Architectural Style provides _____ of design elements	topological constraints of the style	a palette of design elements	Optional semantic Specification	vocabulary
8	Architectural Style define a set of _____ or topological constraints that determine the permitted composition of elements	topological constraints of the style	a palette of design elements	configuration rules	vocabulary
9	An interface that allows external tool to analyze and manipulate_____	description	Multistyle description	architectural description	graphical editor
10	_____ visualization of architectural information together with a graphical editor for manipulating them.	topological constraints of the style	Optional semantic Specification	a palette of design elements	multiple Style Specification

11	_____ is a relation between internal players and Players of the composite	Abstraction mapping	Composition Instructions	part List	Both A and B
12	The Combination of localized definitions and higher level elements makes it possible to _____ for styles	Unformalize rule	formalize rules	formula rule	None of these
13	Aesop adopts a _____ approach	generative	structured	functional	rule based
14	_____ data type rely on an abstraction.	Abstract	shared data	Procedure call	Module
15	Abstraction mapping will be required for _____.	composite connectors	implicit connectors	Explicit Connectors	Instruction Connectors
16	A parts list are _____.	roles and Players	Players and internal Players	Connection and functions	Components and Connectors
17	_____ association between roles and Players.	Connection and functions	Composition Instructions	Abstract mapping	None of these
18	Composition instructions is between _____.	roles and Players	Players and internal Players	Connection and functions	Components and Connectors
19	_____ is relation between internal players and players of composite	Abstraction mapping	composite	Connection and functions	Composition Instructions
20	_____supports component and connectors, style and association of component players	UniCon	RMA	RPC	Filters
21	Architectural styles is composed of which of the following?	A set of component	A topological layout	A set of semantic constraints	All the above
22	Which architectural style's goal is to achieve Integrability?	Data Flow Architecture	Call and Return Architecture	Data Centered Architectures	None of these
23	Which architectural style's goal is to achieve Modifiability with Scalability?	Data Flow Architecture	Call and Return Architecture	Virtual Machine Architecture	None of the mentioned
24	Which architectural style's goal is to achieve Portability?	Data Flow Architecture	Call and Return Architecture	Virtual Machine Architecture	None of the mentioned

25	Which architectural style's goal is to achieve Modifiability with Reuse?	Data Flow Architecture	Call and Return Architecture	Virtual Machine Architecture	None of the mentioned
26	Data Centered architecture is subdivided into which of the following subtypes?	Repository and Blackboard	Batch Sequential, Pipes and Filters	All of the mentioned	None of the mentioned
27	Which of the architectural style is further subdivided into Batch sequential and Pipes & filters?	Data Flow Architecture	Call and Return Architecture	Data Centered Architectures	None of the mentioned
28	Which of the following are types of Call and return architecture?	Main program and subroutine	Remote Procedure Call	Object Oriented	All the above
29	Which of the following type has the main goal to achieve performance?	Main program and subroutine	Remote Procedure Call	Object Oriented	All the above
30	_____ adds a port to a component	editport	extendport	attachport	addport
31	In which of the following style new clients can be added easily?	Data Flow Architecture	Call and Return Architecture	Data Centered Architectures	None of the mentioned
32	_____ are compositions in which code elements are connected in a particular way	components	operators	patterns	closure
33	_____ are conditions in which composition can serve as a subsystem in development of larger systems	components	operators	patterns	closure
34	_____ defines not only functionality but also of performance, fault tolerance and so on.	specification	operators	patterns	closure
35	_____ are module level elements	components	operators	patterns	closure
36	What are the advantage of pipe & filters?	the environment in limited ways	simplify systems maintenance	Interactive applications	All the above

37	What are Virtual Machine Styles?	software on which it is implemented	emphasize on incremental transformation	Interactive applications	None of the mentioned
38	In which of the architectural style the main program is divided into small pieces to achieve modifiability?	Main program and subroutine Architecture	Remote Procedure Call system	Object Oriented or abstract data type system	All of the mentioned
39	In which of the architecture style main program and subroutine systems are decomposed into parts that live on computers connected via a network?	Main program and subroutine Architecture	Remote Procedure Call system	Object Oriented or abstract data type system	All of the mentioned
40	Architectural descriptions treat software systems as composition of _____	components	elements	connectors	modules
41	_____ are simple input/output relations, no retained state.	Computation	memory	Manager	Controller
42	_____ are shared collection of persistent structured data	Computation	memory	Manager	Controller
43	_____ are state and closely related operations	Computation	memory	Manager	Controller
44	_____ governs time sequences of other events.	Computation	memory	Manager	Controller
45	_____ transmits information between entities.	Computation	link	Manager	Controller
46	_____ is a single thread of control passes among definitions	Procedure call	Dataflow	Implicit invocation	Message passing
47	_____ are independent processes interact through streams of data.	Procedure call	Dataflow	Implicit invocation	Message passing
48	_____ is a computation that is invoked by the occurrence of an event.	Procedure call	Dataflow	Implicit invocation	Message passing
49	Independent processes interact by explicit _____	Procedure call	Dataflow	Implicit invocation	Message passing
50	_____ are components that operate concurrently on the same data space.	Instantiation	Shared Data	Components	Operators

51	_____ are Instantiators that uses capabilities instantiated definition by providing space required by instance.	Instantiation	Shared Data	Components	Operators
52	_____ are primitive semantic elements and their values.	Instantiation	Shared Data	Components	Operators
53	_____ are functions that combine components	Instantiation	Shared Data	Components	Operators
54	_____ are rules for naming expressions of components and operators	Abstraction	closure	Specification	Components
55	_____ are rules to determine which abstractions can be added to the classes of primitive components and operators.	Abstraction	closure	Specification	Components
56	_____ are association of semantics to the syntactic forms arrays and so on	Abstraction	closure	Specification	Components
57	_____ capabilities allow us to combine independent architecture elements in to larger systems.	Composition	Abstraction	Reusability	Configuration
58	_____ describe the components and their interactions within software architecture.	Composition	Abstraction	Reusability	Configuration
59	It should be possible to reuse components, connectors, and architectural patterns in different architectural descriptions.	Composition	Abstraction	Reusability	Configuration
60	It should be possible to combine multiple heterogeneous architectural descriptions.	Heterogeneity	Analysis	Reusability	Configuration

ANSWER
Aesop
Idioms and Patterns
Localized Pattern
Reference Model
Communication Reference Model
4
vocabulary
configuration rules
architectural description
multiple Style Specification

Abstraction mapping
formalize rules
generative
Abstract
composite connectors
Components and Connectors
Composition Instructions
roles and Players
Both A and B
UniCon
A set of semantic constraints
Data Centered Architectures
Call and Return Architecture
Virtual Machine Architecture

Data Flow Architecture
Repository and Blackboard
Data Flow Architecture
All the above
Remote Procedure Call
addport
Data Centered Architectures
patterns
closure
specification
components
the environment in limited ways

software on which it is implemented
Main program and subroutine Architecture
Remote Procedure Call system
components
Computation
memory
Manager
Controller
link
Procedure call
Dataflow
Implicit invocation
Message passing
Shared Data

Instantiation
Components
Operators
Abstraction
closure
Specification
Composition
Abstraction
Reusability
Heterogeneity

Register Number_____

[17CSP204]

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed University Established Under Section 3 of UGC Act 1956)
Coimbatore - 641021.
(For the candidates admitted from 2017 onwards)

M.Sc DEGREE EXAMINATION

COMPUTER SCIENCE

FIRST INTERNAL EXAMINATION

INTRODUCTION TO SOFTWARE ARCHITECTURE

Class : I M.Sc CS **Duration : 2 Hours**
Date & Session : 01.02.18 & FN **Maximum : 50 Marks**

PART-A (20 X 1 = 20 Marks)
(Answer ALL the Questions)

1. _____ is emerging as an important discipline for engineers of software.
a) Software Architecture b) Software Engineering
c) Software Design d) Software Development

2. As the size and complexity of software increase _____ become significant.
a) Choice of Algorithm b) Data Structures
c) Specification of overall system structure d) None of these

3. In _____ style each component has a set of inputs and a set of outputs.
a) Pipes and filters b) pipes c) filter d) streams

4. _____ restrict the amount of data that can reside on a pipe.
a) pipelines **b) bounded pipes** c) typed pipes d) filter

5. _____ is a separate, independent parcels of application dependent knowledge.
a) blackboard data structure b) Control **c) The knowledge sources** d) virtual machine

6. Process variable whose value can be changed by the controller.
a) input **b) Manipulated variable** c) controlled d) open loop

7. Detailed understanding of software architectures allows the engineer to make principled choices among _____.
a) design alternatives b) high-level relationships c) components d) interfaces

8. _____is a repository which activates external components through implicit invocation.
a) active database b) database c) queue d) style

9. The KWIC means
a) Key Word in Context b) Key word in Constant
c) Key word as Context d) Key word as Capital

10. A _____system exists to maintain the speed of a car, even over varying terrain.
a) System on/off **b) cruise control** c) Engine on/off d) Increase/Decrease Speed

11. An _____ is an instrumentation system that samples electrical signals and displays pictures of them on a screen.
a) traces b) pulses c) KWIC **d) oscilloscope**

12. A _____system is one that controls a manned or partially manned vehicle.
a) KWIC b) oscilloscopes c) cruise control **d) Mobile Robotics**

13. _____ transformers convert the waveforms into visual data.
a) Display transformers b) Signal transformers
c) Acquisition transformers d) Trigger transformers

14. _____ systems provide a means of codifying the problem-solving skills of human experts.
a) knowledge base **b) Rule based systems** c) cruise control d) oscilloscope

15. The first major blackboard system was the _____
a) HEARSAY-II speech recognition system b) Rule based systems
c) Knowledge sources d) cruise control

16. The TCA architecture is based on hierarchy of _____.
a) Task trees b) exception handlers c) Wiretap d) Message

17. The earliest shared information systems consisted of separate programs for separate _____.
a) subtasks b) main task c) components d) modules

18. In Yourdon data flow diagram, Processes are depicted as _____
a) Computer **b) circles, or bubbles** c) Boxes d) Square

19. When requirements for interaction appear, new organizations allowed independent processing subsystems to interact through_____ .
a) a shared datastore b) pipes c) filters d) components

20. A massive _____ got the transactions into the same order as the records on the sequential master file.
a) transaction sort b) edit program c) update programs d) Data Processing

PART-B (3 X 2 = 6 Marks)
(Answer ALL the Questions)

21. Define Software Architecture.

Software architecture involves the description of elements from which systems are built, interactions among these elements, patterns that guide their composition, and constraints on these patterns.

A particular system is defined as a collection of components and interactions among those components. Such a system may be used as a (composite) element in a larger system.

Components: clients & servers, databases, filters, and layers in a hierarchical system.

Interactions at this level of design can be simple and familiar. Ex: procedure calls, shared variable access.

22. Write about Blackboard architectural style.
Blackboard:

Three major parts:

- **Knowledge sources**: Separate, independent parcels of application dependents knowledge.
- **Blackboard data structure:** Problem solving state data, organized into an application dependent hierarchy
- **Control:** Driven entirely by the state of blackboard; Invocation of a knowledge source (ks) is triggered by the state of blackboard.

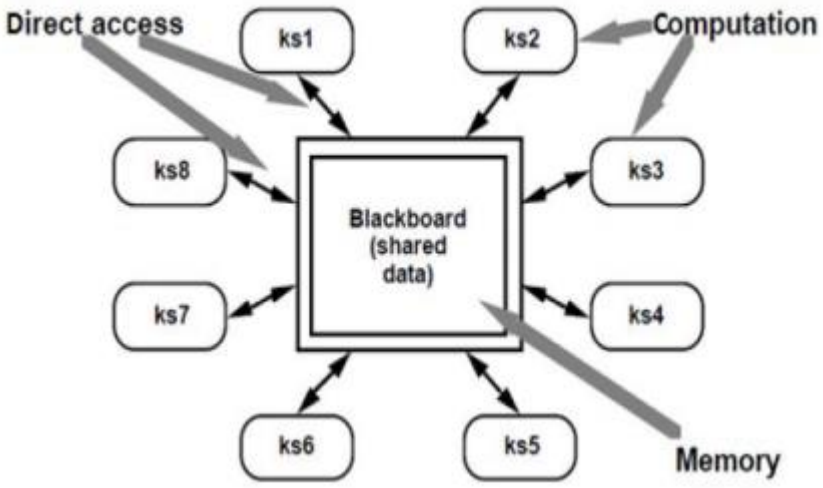


Fig: The Blackboard

23. Write short notes on KWIC system.

- KWIC index system accepts an ordered set of lines.
- Each line is an ordered set of words and each word is an ordered set of characters.
- Any line may be circularly shifted by repeated removing the first word and appending it at the end of the line.
- KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

PART-C (3 X 8 = 24 Marks)
(Answer ALL the Questions)

24. a. Elucidate about An Engineering Discipline for Software.

Software engineering is a label applied to a set of current practices for software development. The more common usage refers to the disciplined application of scientific knowledge to resolve conflicting constraints and requirements for problems of immediate, practical significance.

Definitions of engineering share some common clauses

Creating cost-effective solutions…..

.... to practical problems

…. by applying scientific knowledge...

…. building things...

…. in the service of mankind

Routine and Innovative Design

Engineering design tasks are of several kinds; one of the most significant distinctions among them separates routine from innovative design.

Routine Design: involves solving familiar problems, reusing large portions of prior solutions.

Innovative Design: It involves finding novel solutions to unfamiliar problems. One path to increased productivity is identifying applications that could be routine and¬ developing appropriate support.

A Model for the Evolution of Engineering Discipline

Engineering has emerged from ad hoc practice in two stages.

1. Management and production techniques enable routine production.

2. The problems of routine production stimulate the development of a supporting science; the mature science eventually merges with established practice to yield professional engineering practice.

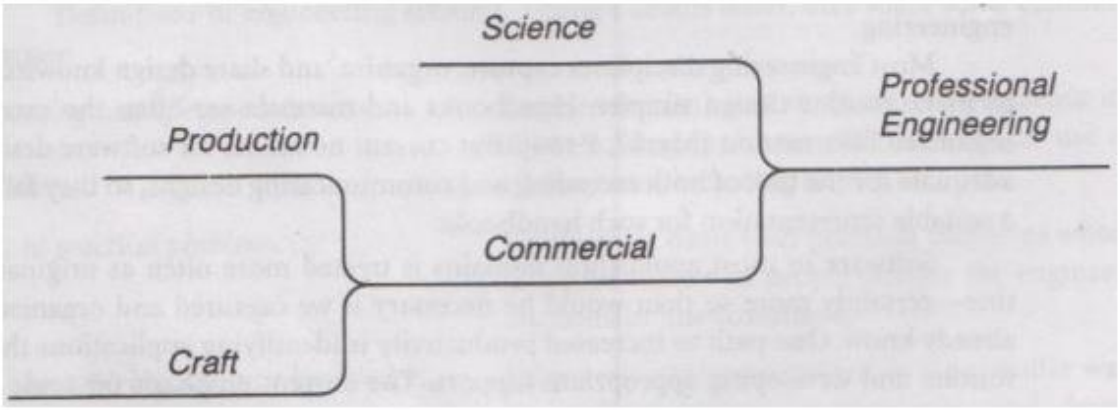


Fig: Evolution of an Engineering Discipline

The lower lines track the technology, and the upper lines show how the entry of production skills and scientific knowledge contribute new capability to the engineering practice.

The Current State of Software Technology

The engineering problem is creating cost-effective solutions to practical problems... building things in the service of mankind.

Scientific basis for Engineering practice

Engineering practice emerges from commercial practice by exploiting the results of a companion science.

One characterization of progress in programming languages and tools has been regular increases in abstraction level—or the conceptual size of software designers building blocks. To place the field of Software Architecture into perspective let us begin by looking at the historical development of abstraction techniques in computer science.

Maturity of supporting science:

- Each program contains algorithms and data structures.
- Algorithms and data structures began to be abstracted from individual programs.

Research on abstract data types dealt with such issues as the following:

- Specifications (abstract models, algebraic axioms)

- Software structure (bundling representation with algorithms)

- Language issues (modules, scope, user-defined types)

- Information hiding (protecting integrity of information not in specification)

- Integrity constraints (invariants of data structures)

- Rules for composition (declarations)

Abstract Data Types

In the late 1960s, good programmers shared an intuition about software development: If you get the data structures right, the effort will make development of the rest of the program

much easier. The abstract data type work of the 1970s can be viewed as a development effort that converted this intuition into a real theory. The conversion from an intuition to a theory involved understanding

• *the software structure* (which included a representation packaged with its primitive operators),

• *specifications* (mathematically expressed as abstract models or algebraic axioms),

• *language issues* (modules, scope, user-defined types),

• *integrity of the result* (invariants of data structures and protection from other manipulation),

• *rules for combining types* (declarations),

• *information hiding* (protection of properties not explicitly included in specifications).

The effect of this work was to raise the design level of certain elements of software systems, namely abstract data types, above the level of programming language statements or individual algorithms. This form of abstraction led to an understanding of a good organization for an entire module that serves one particular purpose. This involved combining representations, algorithms, specifications, and functional interfaces in uniform ways. Certain support was required from the programming language, of course, but the abstract data type paradigm allowed some parts of systems to be developed from a vocabulary of data types rather than from a vocabulary of programming-language constructs.

Interaction between Science and Engineering:

The development of good models within the software domain follows the pattern of following figure.

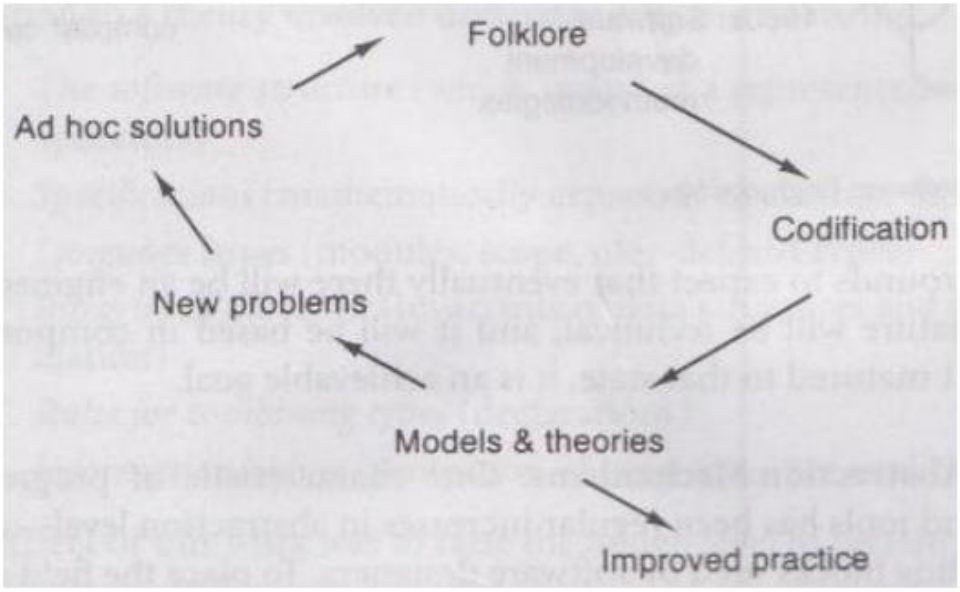


Fig: Codification Cycle for Science and Engineering

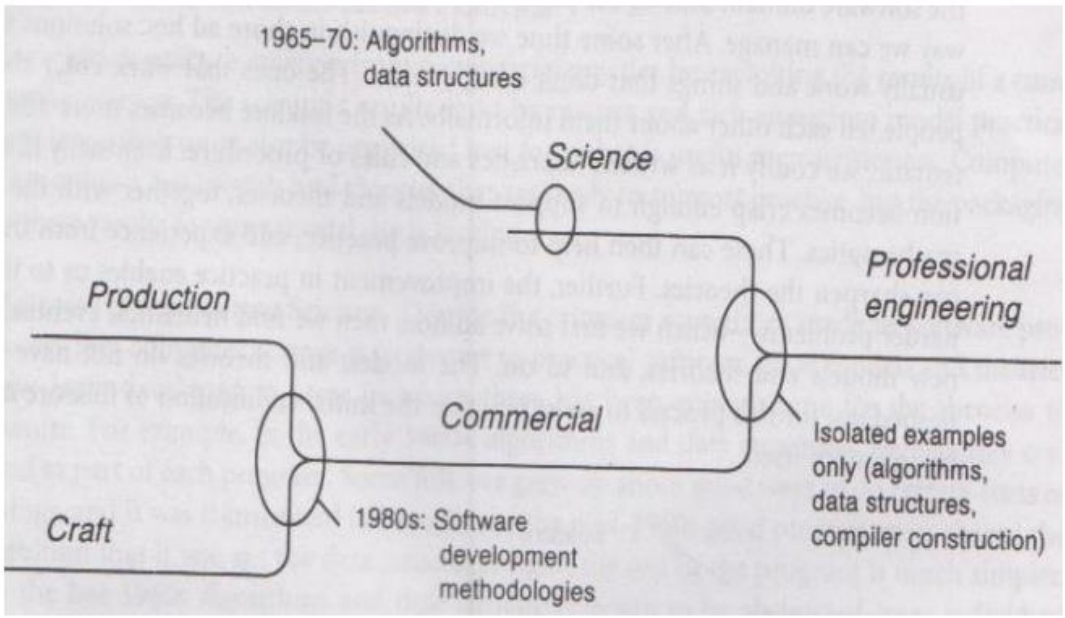


Fig: Evolution of Software Engineering

Codification through Abstraction Mechanisms:

One characteristic of progress in programming languages and tools has been regular increases in abstraction level—or the conceptual size of the building blocks used by software designers.

The conversion from an intuition to a theory involved understanding the following:

The software structure, Specifications, Language issues, Integrity of the result, Rules for combining types, Information hiding.

(OR)

b. Describe about the architectural style Pipes and Filters and Layered Systems with example.

PIPES AND FILTERS:

- Each components has set of inputs and set of outputs
- A component reads streams of data on its input and produces streams of data on its output.
- By applying local transformation to the input streams and computing incrementally, so that output begins before input is consumed. Hence, components are termed as filters.
- Connectors of this style serve as conducts for the streams transmitting outputs of one filter to inputs of another. Hence, connectors are termed pipes.

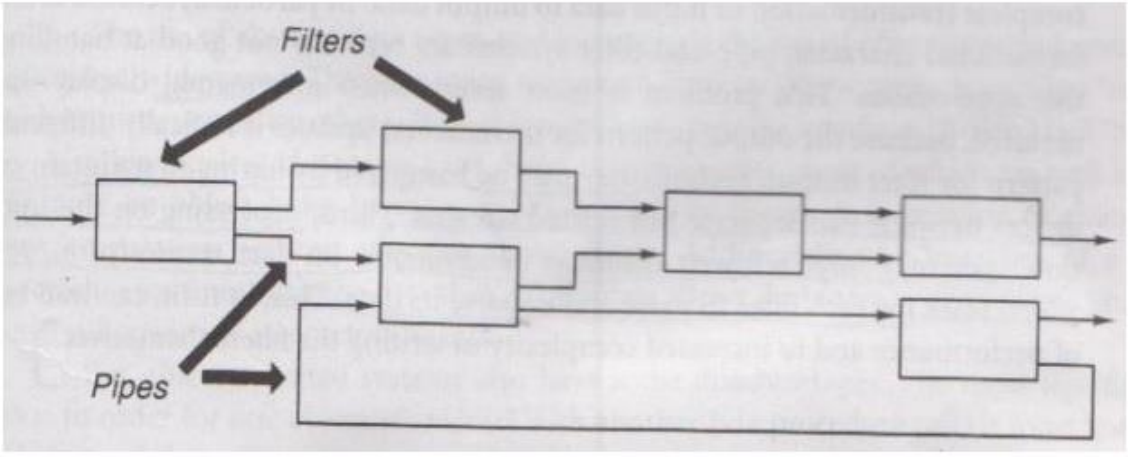


Fig : Pipes and Filters

Conditions (invariants) of this style are:

- Filters must be independent entities.
- They should not share state with other filter
- Filters do not know the identity of their upstream and downstream filters.
- Specification might restrict what appears on input pipes and the result that appears on the output pipes.
- Correctness of the output of a pipe-and-filter network should not depend on the order in which filter perform their processing.

Common specialization of this style includes:

- Pipelines: Restrict the topologies to linear sequences of filters.
- Bounded pipes: Restrict the amount of data that can reside on pipe.
- Typed pipes: Requires that the data passed between two filters have a well-defined type.

Advantages:

- They allow the designer to understand the overall input/output behavior of a system as a simple composition of the behavior of the individual filters
- They support reuse: Any two filters can be hooked together if they agree on data.
- Systems are easy to maintain and enhance: New filters can be added to exciting systems.
- They permit certain kinds of specialized analysis eg: deadlock, throughput
- They support concurrent execution.

Disadvantages:

- They lead to a batch organization of processing.
- Filters are independent even though they process data incrementally.
- Not good at handling interactive applications
    - When incremental display updates are required.
    - They may be hampered by having to maintain correspondences between two separate but related streams.

- Lowest common denominator on data transmission.

- This can lead to both loss of performance and to increased complexity in writing the filters.

LAYERED SYSTEMS:

- A layered system is organized hierarchically, each layer provides service to the layer above it and serving as a client to the layer below.
- Inner layers are hidden from all except the adjacent layers.
- Connectors are defined by the protocols that determine how layers interact with each other.
- Goal is to achieve qualities of modifiability portability.

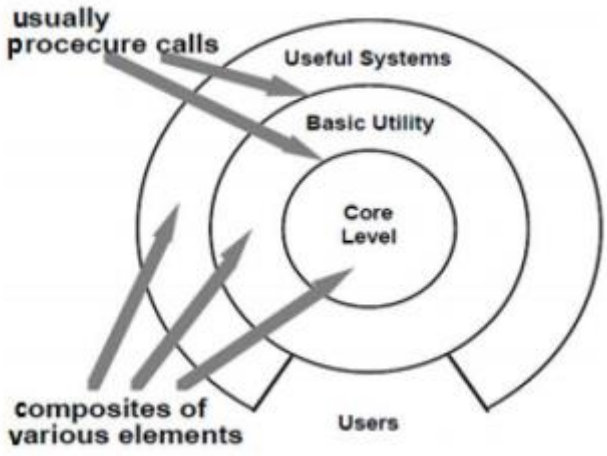


Fig: Layered Systems

Examples:

 Layered communication protocol

 Operating systems

Database systems

Advantages:

- They support designs based on increasing levels abstraction.
- Allows implementers to partition a complex problem into a sequence of incremental steps.
- They support enhancement
- They support reuse.

Disadvantages:

- Not easily all systems can be structures in a layered fashion.
- Performance may require closer coupling between logically high-level functions and their lower-level implementations.
- Difficulty to mapping existing protocols into the ISO framework as many of those
- protocols bridge several layers.

25. a. Discuss about Key Word in Context – (i) Abstract Data Types (ii) Pipe and Filter model with a neat diagram.

Parnas proposed the following problems:

KWIC index system accepts an ordered set of lines. Each line is an ordered set of words and each word is an ordered set of characters. Any line may be circularly shifted by repeated removing the first word and appending it at the end of the line. KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

Parnas used the problem to contrast different criteria for decomposing a system into modules.

He describes 2 solutions:

a) Based on functional decomposition with share access to data representation.

b) Based on decomposition that hides design decision.

From the point of view of Software Architecture, the problem is to illustrate the effect of changes on software design. He shows that different problem decomposition vary greatly in their ability to withstand design changes. The changes that are considered by parnas are:

1. The changes in processing algorithm:

Eg: line shifting can be performed on each line as it is read from input device, on all lines after they are read or an demand when alphabetization requires a new set of shifted lines.

2. Changes in data representation:

Eg: Lines, words, characters can be stored in different ways. Circular shifts can be stored explicitly or implicitly Garlan, Kaiser and Notkin also use KWIC problem to illustrate modularization schemes based on implicit invocation. They considered the following.

3. Enhancement to system function:

Modify the system to eliminate circular shift that starts with certain noise change the system to interactive.

4. Performance:

Both space and time

ABSTRACT DATA TYPES

- Decomposes The System Into A Similar Set Of Five Modules.
- Data is no longer directly shared by the computational components.
- Each module provides an interface that permits other components to access data only by invoking procedures in that interface.

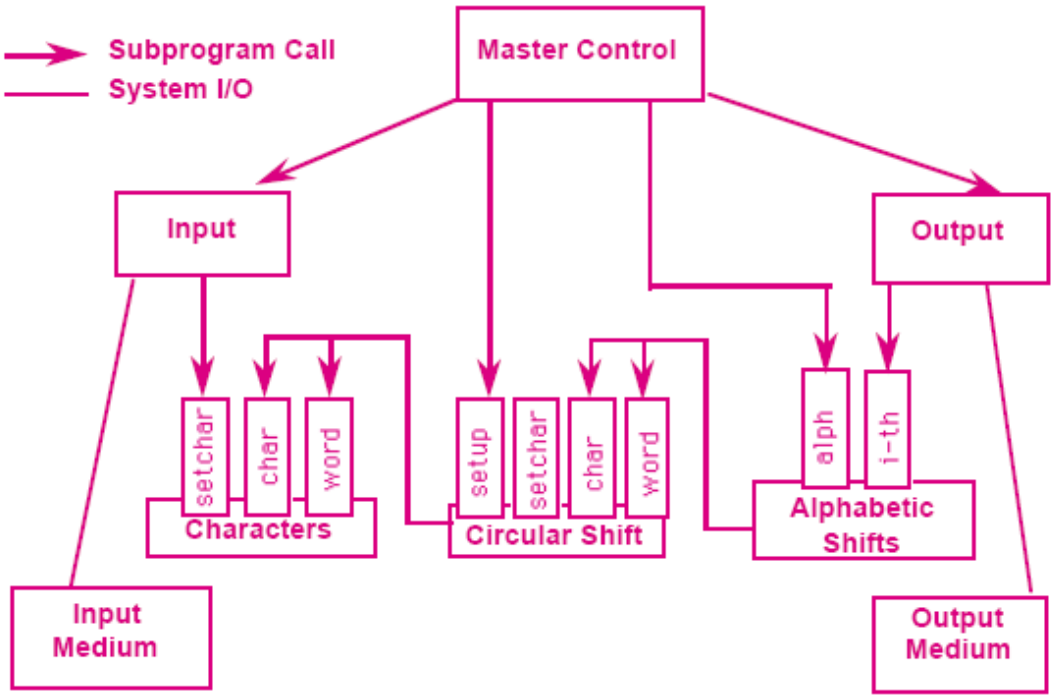


Figure 7: KWIC – Abstract Data Type Solution

Advantage:

- Both Algorithms and data representation can be changed in individual modules without affecting others.
- Reuse is better supported because modules make fewer assumption about the others with which they interact.

Disadvantage:

- Not well suited for functional enhancements
- To add new functions to the system
- To modify the existing modules.

PIPES AND FILTERS:

- Four filters: Input, Output, Shift and alphabetize
- Each filter process the data and sends it to the next filter
- Control is distributed

  o Each filter can run whenever it has data on which to compute.

- Data sharing between filters are strictly limited.

Advantages:

- It maintains initiative flow of processing
- It supports reuse
- New functions can be easily added to the system by inserting filters at appropriate level.
- It is easy to modify.

Disadvantages:

- Impossible to modify the design to support an interactive system.
- Solution uses space inefficiently.

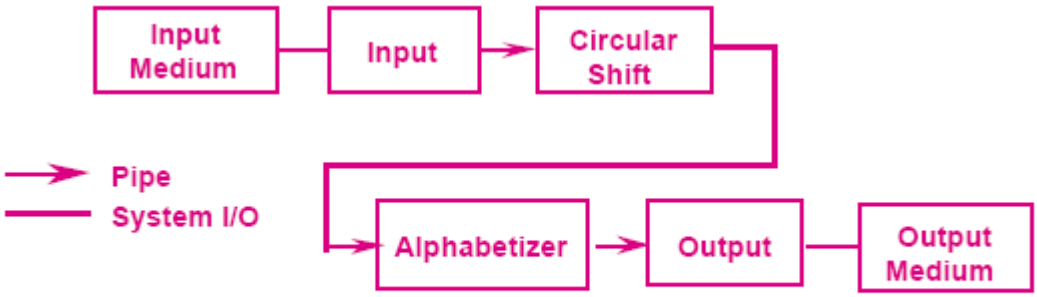


Figure 9: KWIC – Pipe and Filter Solution

(OR)

b. Explain the Cruise Control system in detail.

CRUISE CONTROL

A cruise control (CC) system that exists to maintain the constant vehicle speed even over varying terrain.

Inputs:

System On/Off: If on, maintain speed

Engine On/Off: If on, engine is on. CC is active only in this state

Wheel Pulses: One pulse from every wheel revolution

Accelerator: Indication of how far accelerator is de-pressed

Brake: If on, temp revert cruise control to manual mode

Inc/Dec Speed: If on, increase/decrease maintained speed

Resume Speed: If on, resume last maintained speed

Clock: Timing pulses every millisecond

Outputs:

Throttle: Digital value for engine throttle setting

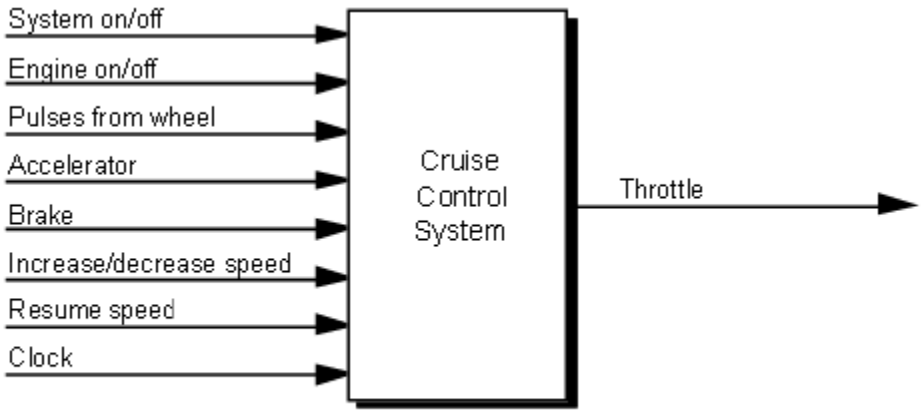


Figure 5: Booch block diagram for cruise control

Restatement of Cruise-Control Problem

Whenever the system is active, determine the desired speed, and control the engine throttle setting to maintain that speed.

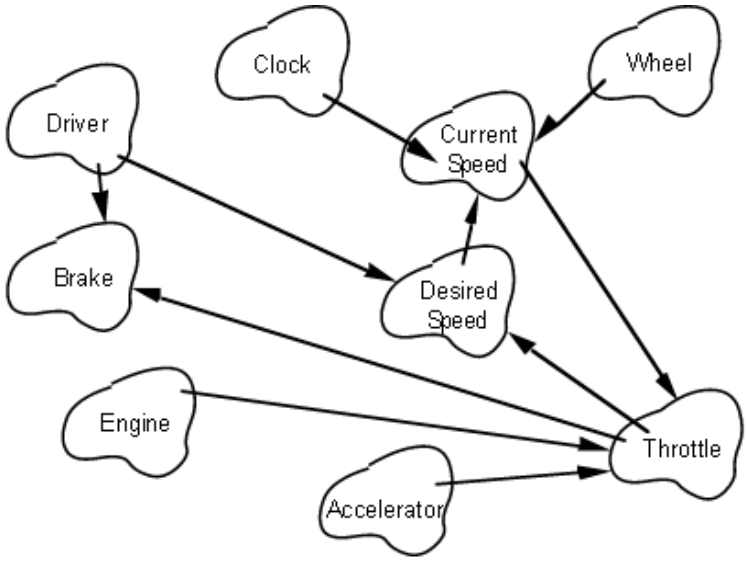
OBJECT VIEW OF CRUISE CONTROL

□□Each element corresponds to important quantities and physical entities in the system

□□Each blob represents objects

□□Each directed line represents dependencies among the objects

The figure corresponds to Booch's object oriented design for cruise control



PROCESS CONTROL VIEW OF CRUISE CONTROL

□□□**Computational Elements**

□□*Process definition* - take throttle setting as I/P & control vehicle speed

□□*Control algorithm* - current speed (wheel pulses) compared to desired speed

o Change throttle setting accordingly presents the issue:

o decide how much to change setting for a given discrepancy

□□Data Elements

□□*Controlled variable:* current speed of vehicle

□□*Manipulated variable:* throttle setting

□□*Set point:* set by accelerator and increase/decrease speed inputs

□□system on/off, engine on/off, brake and resume inputs also have a bearing

□□*Controlled variable sensor*: modelled on data from wheel pulses and clock

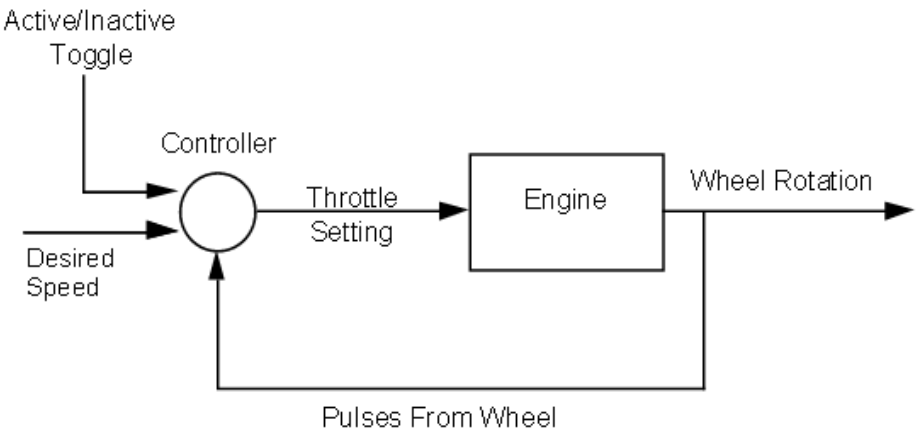


Figure 7: Control Architecture for Cruise Control

Figure 3.18 control architecture for cruise control

The active/inactive toggle is triggered by a variety of events, so a state transition design is natural. It's shown in Figure. The system is completely off whenever the engine is off. Otherwise there are three inactive and one active state.

For simplicity we assume brake application is atomic so other events are blocked when the brake is on. A more detailed analysis of the system states would relax this Assumption.

The active/inactive toggle is triggered by a variety of events, so a state transition design is natural.The system is completely off whenever the engine is off. Otherwise there are three inactive and one active states. In the first inactive state no set point has been established. In the other two, the previous set point must be remembered:

When the driver accelerates to a speed greater than the set point, the manual Accelerator controls the throttle through a direct linkage (note that this is the only use of the accelerator position in this design, and it relies on relative effect rather than absolute position); when the driver uses the brake the control system is inactivated until the resume signal is sent. The active/inactive toggle input of the control system is set to active exactly when this state machine is in state Active

Determining the desired speed is simpler, since it does not require state other than the current value of desired speed (the set point). Any time the system is off, the set point is undefined. Any time the system on signal is given (including when the system is already on) the set point is set to the current speed as modeled by wheel pulses.

The driver also has a control that increases or decreases the set point by a set amount. This, too, can be invoked at any time (define arithmetic on undefined values to yield undefined values). Figure 9 summarizes the events involved in determining the set point. Note that this process requires access to the clock in order to estimate the current speed based on the pulses from the wheel.

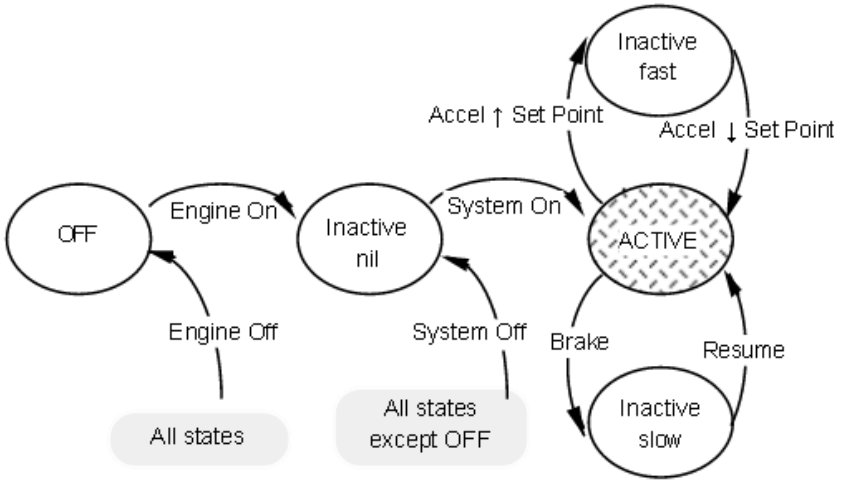


Figure 8: State Machine for Activation

Figure state machine for activation

Event	*Effect on desired speed*
Engine off, system off	Set to "undefined"
System on	Set to current speed as estimated from wheel pulses
Increase speed	Increment desired speed by constant
Decrease speed	Decrement desired speed by constant

Figure 9: Event Table for Determining Set Point

Combine the control architecture, the state machine for activation, and the event table for determining the set point into an entire system.

Compose the control architecture, the state machine for activation, and the event table for determining the set point into an entire system. Although there is no need for the control unit and set point determination to use the same clock, we do so to minimize changes to the original problem statement. Then, since current speed is used in two components, it would be reasonable for the next elaboration of the design to encapsulate that model in a reusable object; this would encapsulate the clock.

All of the objects in Booch's design (Figure 6) have clear roles in the resulting system. It is entirely reasonable to look forward to a design strategy in which the control loop architecture is used for the system as a whole and a number of other architectures, including objects and state machines, are used in the elaborations of the elements of the control loop architecture.

The shift from an object-oriented view to a control view of the cruise control architecture raised a number of design questions that had previously been slighted: The separation of process from control concerns led to explicit choice of the control discipline.

The limitations of the control model also became clear, including possible inaccuracies in the current speed model and incomplete control at high speed. The dataflow character of the model showed irregularities in the way the input was specified, for example mixture of state and event inputs and the inappropriateness of absolute position of the accelerator

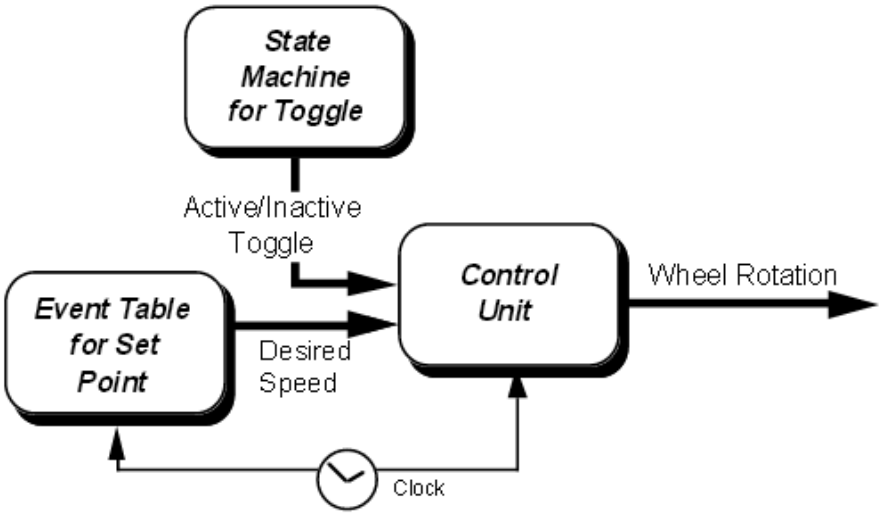


Figure 10: Complete cruise control system

26. a. Write in detail about Database Integration – (i) Batch Sequential (ii) Simple Repository.

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. Interactive demands required individual transactions to complete in real time.

Information began to appear redundantly in multiple databases, and geographic distribution added communication complexity.

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.

As databases became more common, information about a business became distributed among multiple databases. Hence data become inconsistent and incomplete. 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
- multi-databases.

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*: accepts transaction inputs and perform validation without accessing the database.
- a massive *transaction sort*: get 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*: produce periodic reports

Batch sequential architecture:

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.

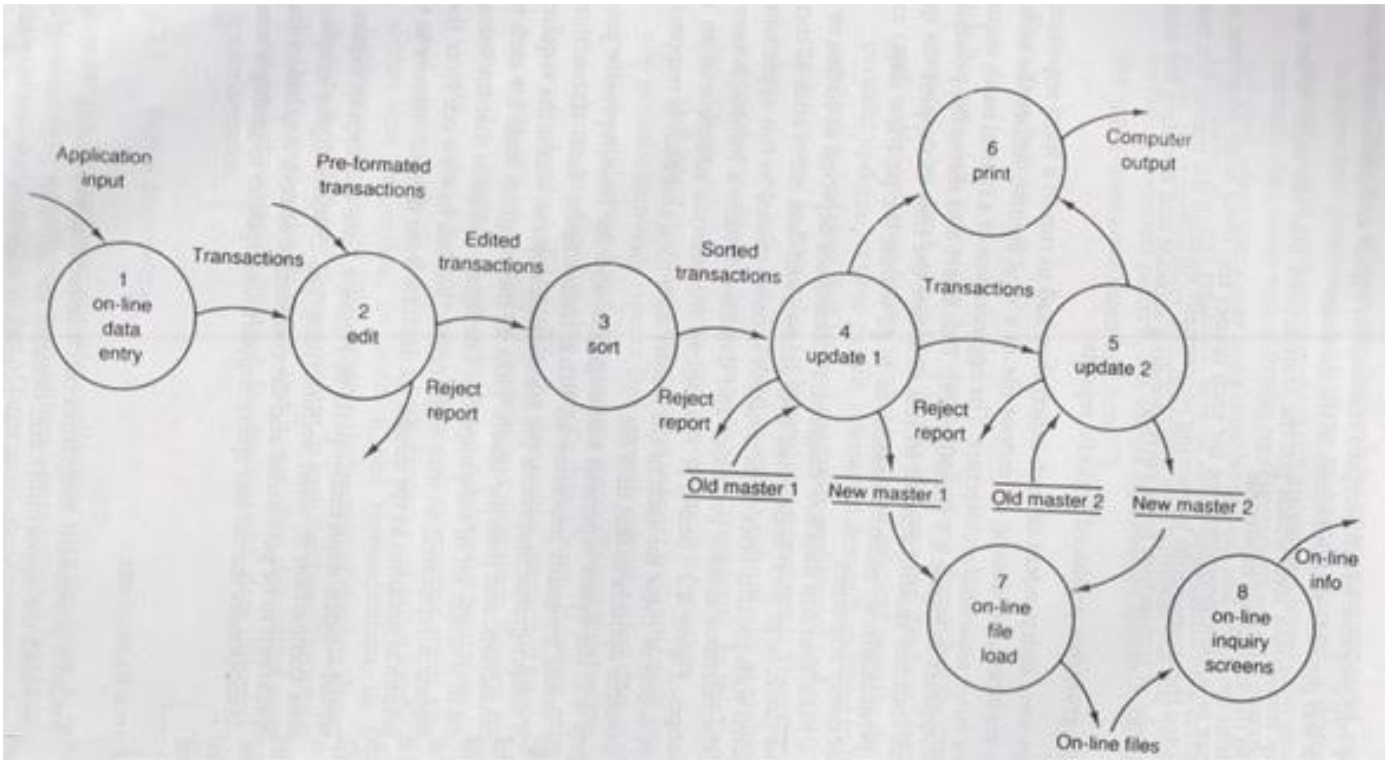


Fig: Data flow diagram for batch databases

The above figure 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.

Above figure 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.

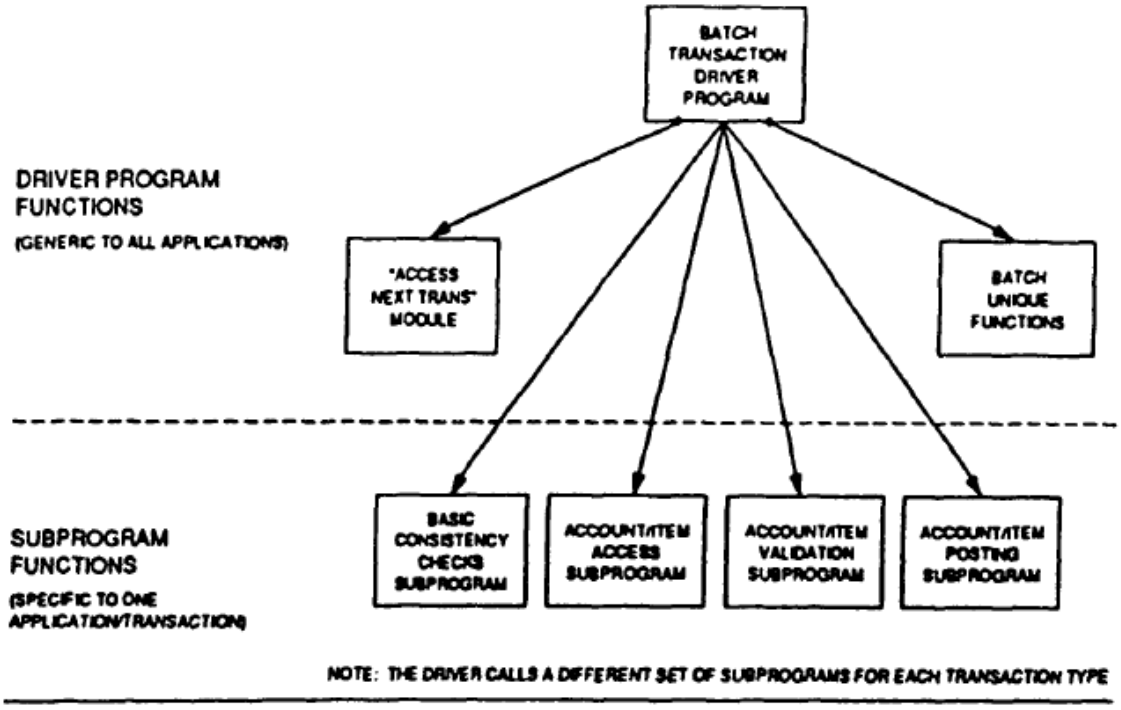


Fig: Internal structure of batch update process

Above figure 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. Here, 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 transaction, and post the result. Thus all the program logic for each transaction is localized in a single set of subprograms.

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.

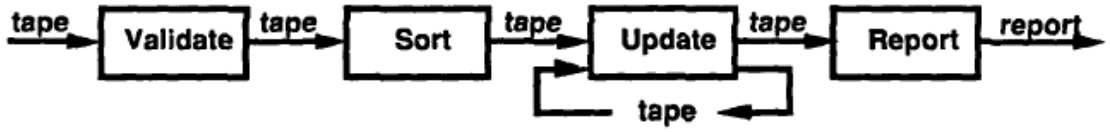


Figure 4: Batch sequential database architecture

Fig: Batch sequential database architecture

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.

 Second, as organizations grew, the set of transactions and queries grew.

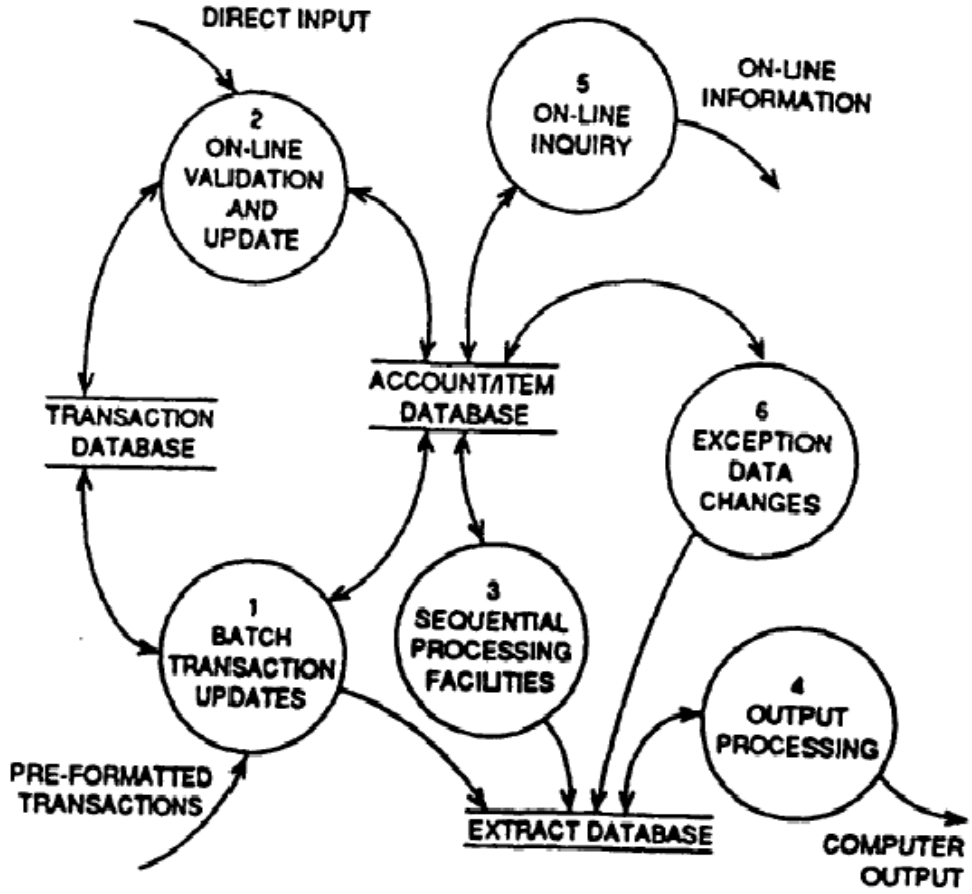


Figure 5: Data fow diagram for interactive database

 Here, 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.

 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.

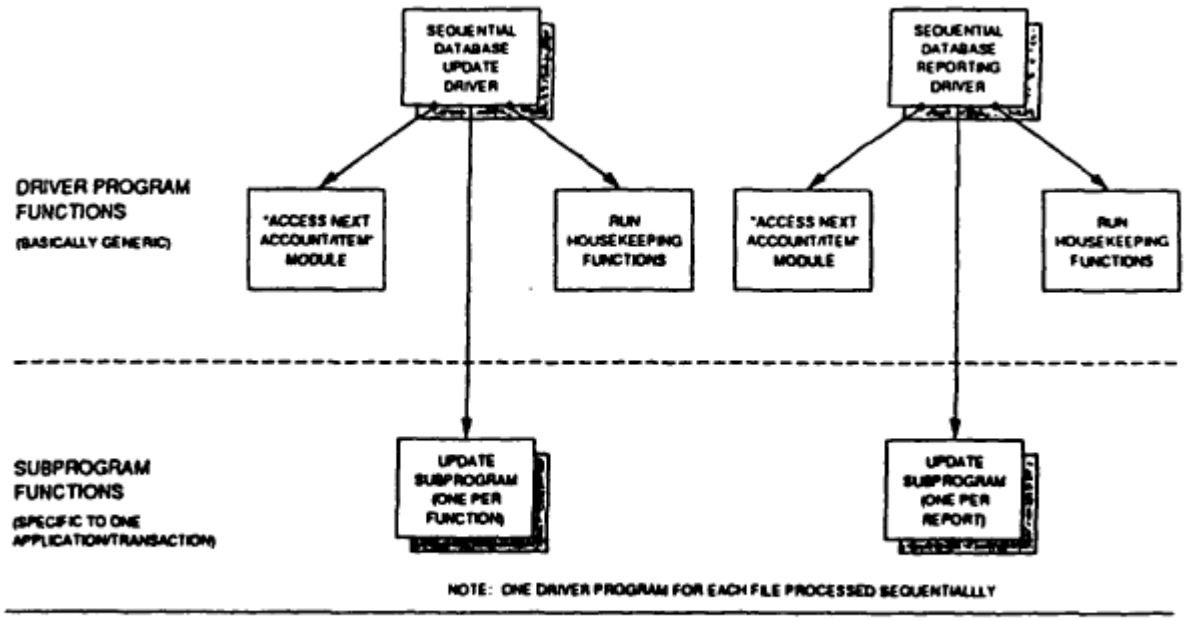


Figure 6: Internal structure of interactive update process

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.

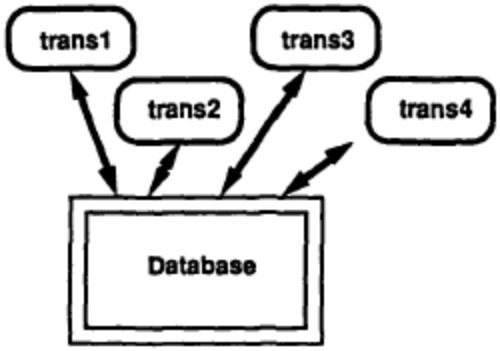


Figure 7: Simple repository database architecture

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.

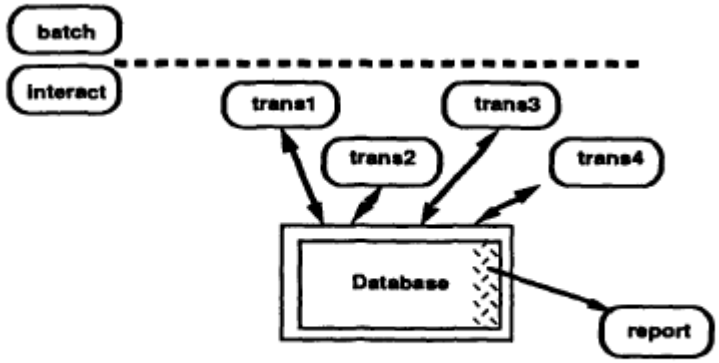


Figure 8: Repository architecture for database showing control and reporting

(OR)

b. Explain in detail about Integration in Software Development Environment – (i) Repository (ii) Transition from Batch Sequential to Repository.

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 repository of the compiler provided a focus for this data collection.
- 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.

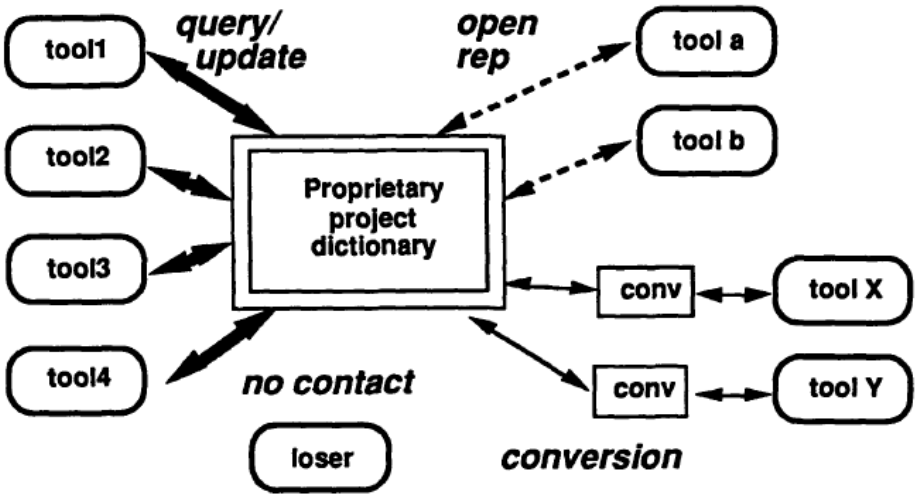


Figure 18: Software tools with shared representation

INTEGRATION IN SOFTWARE DEVELOPMENT ENVIRONMENTS:

 Software development has relied on **software tools** whereas 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.

 Tools now support analysis, configuration control, debugging, testing, and documentation as well.

Transition from Batch Sequential to Repository:

 Our view of the architecture of a system can change in response to improvements in technology.

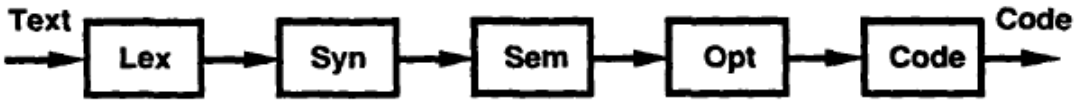


Figure 14: Traditional compiler model

 We often refer to this compilation model as a pipeline, even though it was closer to a batch sequential architecture in which each transformation ("pass") ran to completion before the next one started.

 Symbol Table was not part of the data that flowed from one pass to another but rather existed outside all the passes.

 The algorithms and representations of compilation grew more complex, and increasing attention turned to the intermediate representation of the program during compilation.

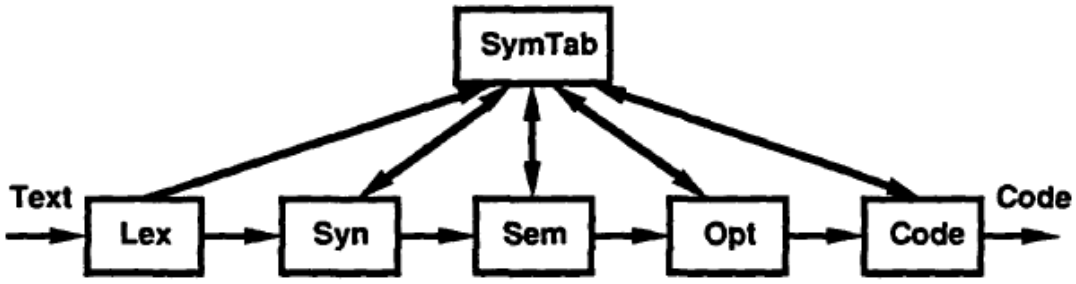


Figure 15: Traditional compiler model with symbol table

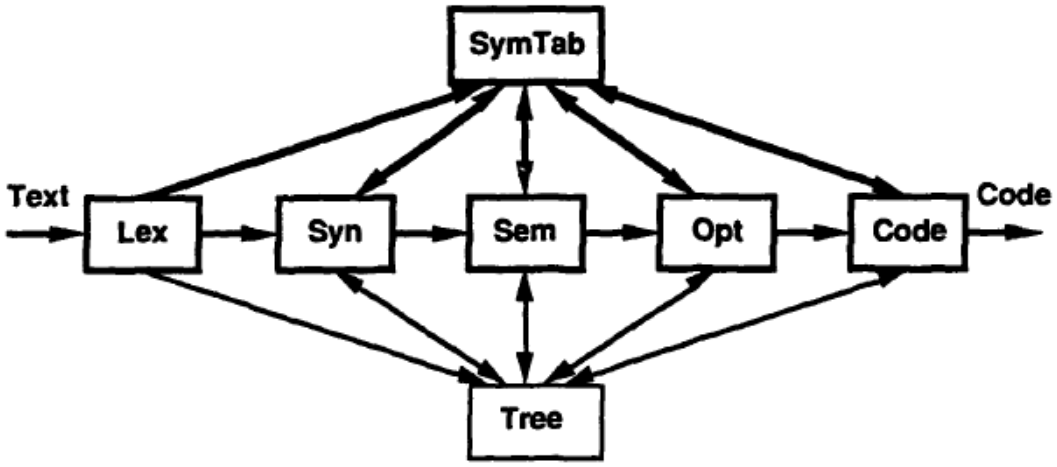


Figure : Modern canonical compiler

 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.

 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*.

 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.

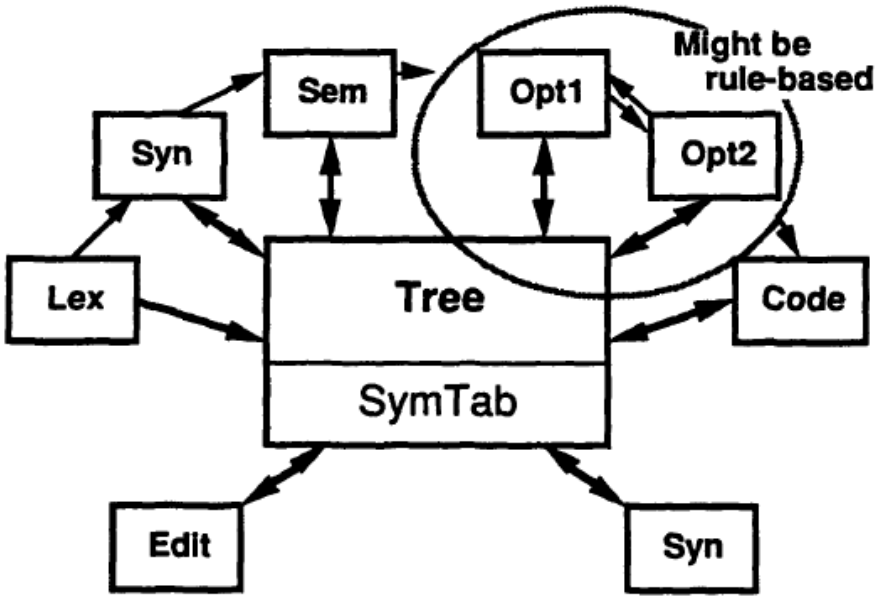


Figure 17: Repository view of modern compiler