Using Architectural Styles in Design

Lecture 6

In this lecture you will learn:

- Use of software architectural styles in design
- Selection of appropriate architectural style
- Combinations of styles
- Case studies
  - Case study 1: Keyword Frequency Vector (KFV)
  - Case study 2: Keyword in Context (KWIC)

Which Architectural Style should be used?

- The selection of software architectural style should take into consideration of two factors:
  - The nature of computation
  - The quality concerns

The selection of software architectural style should take into consideration of two factors:

- The nature of the computation problem to be solved. e.g. the input/output data structure and features, the required functions of the system, etc.
- The quality concerns that the design of the system must take into consideration, such as performance, reliability, security, maintenance, reuse, modification, interoperability, etc.

Characteristic Features of Architectural Styles

- Control Topology
- Data Topology
- Control/Data Interaction Topology
- Control/Data Interaction Direction

Control Topology:

- What geometric form does the control flow for the systems of the style take?
  - E.g. In the main-program-and-subroutine style, components must be organized must be organized into a hierarchical structure.

Data Topology

- Geometric form does the data flow for the systems of the style take?
  - E.g. In the batch sequential processing style, components are organized in a linear sequence structure that data are passed from one component to the next.

Control/Data Interaction Topology:

- Are the topological structures of control topology and data topology substantially isomorphic?
  - E.g. For all systems in the batch sequential processing style, the topological structure from the control-flow point of view is identical to the structure from the data-flow point of view.

Control/Data Interaction Direction:

- If the control and data topologies of the systems of the style are the same, does the control flow in the same directions as the data flows or the opposite?
  - E.g. In the batch sequential processing style, data is passed from one component to another in the same direction that control is passed between the components.

Behavioral Features of Architectural Styles

- Synchronicity
- Data Access Mode
- Data Flow Continuity
- Binding Time

Synchronicity:

- How dependent are the components' action upon each other's control state?
  - **Lockstep –** One component's state implies the other component's state.
  - **Synchronous –** Components synchronies at certain states to make sure they are in the synchronized states at the same time in order to cooperate, but the relationships on other states are not predictable.
  - **Asynchronous –** Components are not synchronized at any state, however the state of a component may affect the other components' behavior.
  - **Opportunistic** – Two components work completely independently from each other in parallel.

Data Access Mode

- How is data made available throughout the system?
  - It can **Passed** from one component to another, such as in a message passing style.
  - It can be a **Shared** by making it available at a place accessible to all sharers.
  - If a component copies the data from a public store, modifies it and then reinsert it back to the public store, the data access mode is **Copy-Out-Copy-In**.
  - In some style, data are **Broadcast** or **Multicast** to specific recipients.

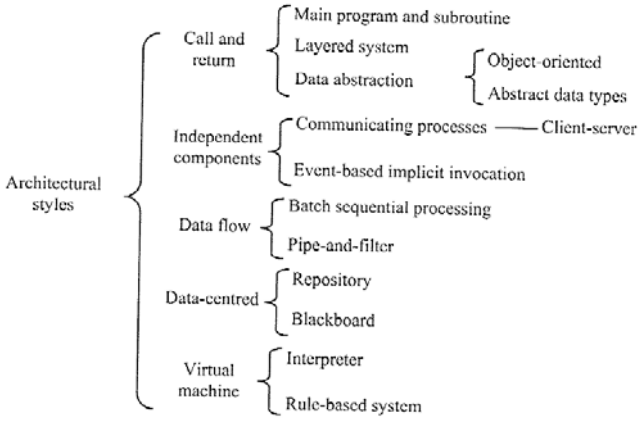
Data Flow Continuity:

- How continuous is the data flow throughout the system?
  - A **Continuous** data flow system has fresh data available all the time.
  - A **Sporadic** data flow system has new data generated at discrete times.
  - Data transfer between components can also be **High Volume** in data intensive systems, or **Low Volume** in computation intensive systems.

Binding Time

- When the names are bound to the entities for control and data transfer. That is, when the identity of a partner in a control or data transfer is established.
  - It can be at **Code-time**, i.e. when the programmer writes the source code.
  - It can be at **Compile-time**, i.e. when the source code is compiled into object code.
  - It can be at **Invocation-time**, i.e. when the operating system initializes the execution of the system.
  - It can be at **Run-time**, during the execution of the system.

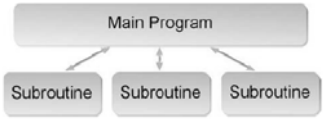
Initial Rules on Selecting An Architecture



- If the problems is decomposable into sequential stages:
  - Then consider data Flow (batch sequential or pipeline)
    - If your problem involves passing rich data representations,:
      - Then avoid pipelines restricted to ASCII.
    - If in addition each stage is incremental, so that later stages can begin before earlier stages finish:
      - Then consider a pipeline architecture.
    - Else If the problem involves transformations on continuous streams of data (or on very long streams):
      - Then consider a pipeline architecture.
- If a central issue is understanding the data of the application, its management, and its representation:
  - Then consider a repository or abstract data type architecture.
    - If the data is long-lived:
      - THEN focus on repositories.
    - If the representation of data is likely to change over the lifetime of the program
      - then define abstract data types in order to confine changes to particular components.
    - If the input data is noisy (low signal-to-noise ratio) and the execution order cannot be predetermined
      - Then consider a blackboard
    - If the execution order is determined by a stream of incoming requests and the data is highly structured
      - Then consider a database management system.
- If your system involves controlling continuing action, is embedded in a physical system, and is subject to unpredictable external perturbation so that preset algorithms go awry:
  - Then consider a closed loop control architecture.
- If you have designed a computation but have no machine on which you can execute it
  - Then consider an interpreter architecture.
    - If your task requires a high degree of flexibility, configurability, loose coupling between tasks, and reactive tasks,
      - Then consider interacting processes.
    - If you have reason not to bind the recipients of signals from their originators:
      - Then consider an event architecture.
    - If the tasks are of a hierarchical nature:
      - Then consider replicated worker or heartbeat style.
    - If the tasks are divided between producers and consumers:
      - Then consider client/server.
    - If it makes sense for all of the tasks to communicate with each other in a fully connected graph:
      - Then consider a token passing style.

Main Program and Subroutine

- In a Call and Return system, a group of subroutines that share a common data store can be grouped together to form a Module.



Purpose

The Main Program and Subroutine architecture supports system modifiability, scalability, and performance.

Motivation

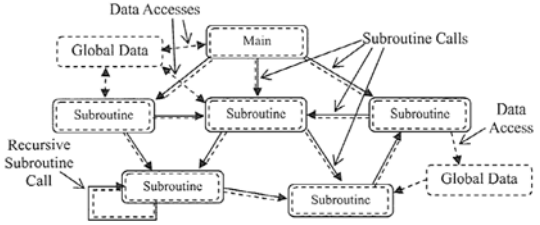
The Main Program and Subroutine is the classic programming pattern. By separating functionality into modules (subroutines), the architecture separates concerns into smaller amounts of complexity, thereby managing the complexity more effectively.

Application

In the Main and Subroutine architecture, there is typically a single thread of control and each component in the hierarchy gets this control from its parent and passes it along to its children. Remote procedure call systems are Main and Subroutine systems that are decomposed into parts that live on computers connected via a network. The actual assignment of parts to processors is deferred until runtime.

Main-Program-and-Subroutine with Shared Data

- A system in Call and Return style can have any topological structure that links subroutines by subroutine calls.
- A common practice is to organize the connections between subroutine in certain patterns and to pack a number of interrelated subroutines into program units. This results in a number of sub-types of the style



7

A Call and Return Architecture with a hierarchical structure with shared data is often called the Main-Program-and-Subroutine with Shared Data.

Main Program and Subroutine

Advantages

- Easily modified
- Grow system functionality by adding more modules
- Simple to analyze control flow

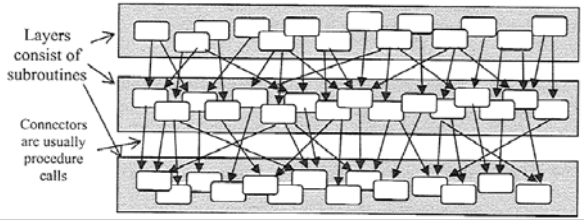
Disadvantages

- Parallel processing may be difficult
- May be difficult to distribute across machines (traditional)
- Exceptions to normal operation are awkward to handle

8

Layered Structure Style

- A layered system is organized hierarchically with each layer providing service to the layer above it and serving as a client to the layer below.
- In some systems inner layers are hidden from all except the adjacent outer layer.
- Example: OSI Model



9

Purpose

The Layered architecture supports system modifiability and portability.

Motivation

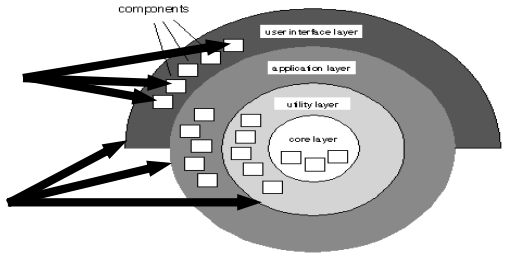
Layered systems are composed of components that are assigned to layers, which control the inter-component interaction. In the pure version of this pattern, each level communicates only with its immediate neighbors.

Application

The lowest layer in the Layered architecture, provides some core functionality, such as hardware or an operating system kernel. Each successive layer is built on its predecessor, hiding the lower layer and providing some services that the upper layers make use of.

Component
Layers - Composites of various elements

Connectors
Usually procedure calls



Layered Structure Style

Advantages	Disadvantages
■ Easily ported to different platforms ■ Easily modified by substituting one layer for a new one	■ Performance degradation due to the communication between layers ■ Structuring systems may be difficult

 10

Advantages:

- Layered systems support designs based on increasing levels of abstraction.
- Complex problems may be partitioned into a series of steps.
- Enhancement is supported through limiting the number of other layers with which communication occurs.
- Support reuse

Disadvantages:

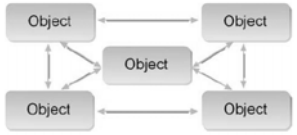
- Disadvantages include the difficulty in structuring some systems into a layers.
- Performance considerations may not be well served by layered systems especially when high level functions require close coupling to low level implementations.
- It may be difficult to find the right level of abstraction especially if existing systems cross several layers.

Issues:

- Constraints include limiting interactions to adjacent layers.
- No one agrees exactly on what a 'layer' is.
  - For example, some layering schemes have very little relationship to the runtime.
  - Others confuse layers with a 'tiered' runtime architecture where the various layers are not only split by build-time packaging, but by running in different processes and possibly different nodes at run time.

Object Oriented

- Data representations and their associated operations encapsulated in an abstract data type.
- The components are the objects and connectors operate through procedure calls (methods).
- Objects maintain the integrity of a resource and the representation is hidden from others.



Purpose

The Object Oriented architecture supports system modifiability, reuse, scalability, and performance.

Motivation

The Object Oriented pattern emphasizes the bundling of data and the knowledge of how to manipulate and access that data. This bundle is an encapsulation that hides its internal secrets from its environment. Access to the object is allowed only through provided operations (or methods).

Application

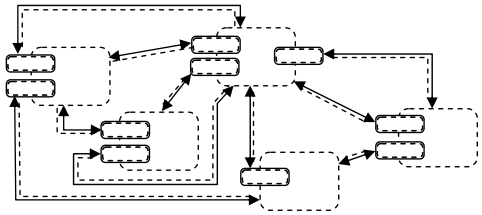
The Object Oriented architecture is more conducive to multi-threaded applications, though traditional implementations are single threaded. The encapsulation promotes reuse and modifiability, principally because it promotes separation of concerns. Well organized Java, C#, and SmallTalk programs are examples of systems using this architecture pattern.

Components:

- Objects, instances of abstract data types

Connectors:

- Operator (methods) calls



Object Oriented

Advantages

- Hierarchical sharing of definitions and code (inheritance)
- Ability to determine the semantics of an operation at runtime (polymorphism)
- Encapsulation
- Loose coupling
- Structure of system is relatively easy to understand

Disadvantages

- More system overhead used to maintain objects
- Must explicitly reference the name and interface of other objects

Advantages

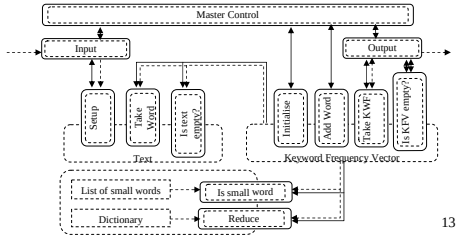
- Modifiability: An object hides its representation from its clients, hence it is possible to change the implementation without affecting those clients
- Structuredness: It decomposes problems into collections of interacting agents

Disadvantages

- Naming problem: An object must know the identity of that other object, Whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it
- Side-effect: 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

Abstract Data Type

- 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.



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.

Abstract Data Type

Advantages

- Server is able to change an implementation without affecting the client.
- Grouping of methods with objects allows for more modular design and therefore decomposes the problems into a series of collections of interacting agents

Disadvantages

- For one object to interact with another, the client object must know how to interact with the server object and therefore must know the identity of the server

Advantages:

- Server is able to change an implementation without affecting the client.
- Grouping of methods with objects allows for more modular design and therefore decomposes the problems into a series of collections of interacting agents.

Disadvantages:

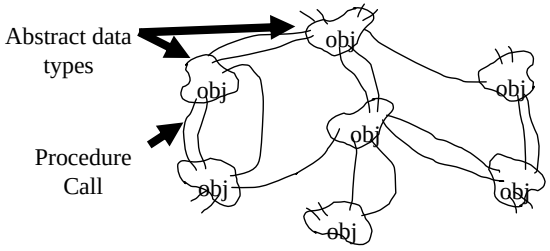
- For one object to interact with another, the client object must know how to interact with the server object and therefore must know the identity of the server.
- If the server changes its interface ALL interacting clients must also change.
  - Services declared as PUBLIC and IMPORTED into the CLIENT
- Also need to consider side affects, A uses B , B uses C and we mod C

Components:

- Objects, instances of abstract data types

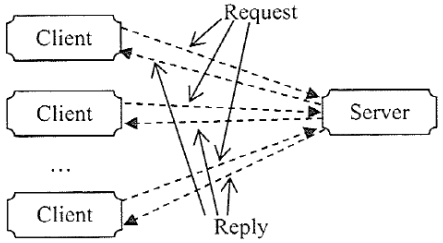
Connectors:

- Operator (methods) calls



Communicating Processes – Client/Server

- All components of a system in the independent component style must be executable.
- They are often called **Processes** if the component is active. Otherwise, they are called **Module**.



15

Purpose

The Communicating Processes architecture supports system modifiability and scalability.

Motivation

The Communicating Processes pattern consists of a number of independent processes or objects that communicate through messages. They send data to each other but typically do not directly control each other. Generally, the messages are passed through named participants.

Application

The Communicating Processes pattern is the classic multi-processing system. A good example of this pattern is the client-server model. A server exists to serve data to one or more clients, which are typically located across a network. The client originates a call to the server, which works, synchronously or asynchronously, to service the client's request. If the server works synchronously, it returns control to the client at the same time that it returns the requested data. If the server works asynchronously, it returns only data to the client, which maintains its own thread of control.

Communicating Processes – Client/Server

Advantages

- Distribution is straightforward
- Easily implemented in parallel

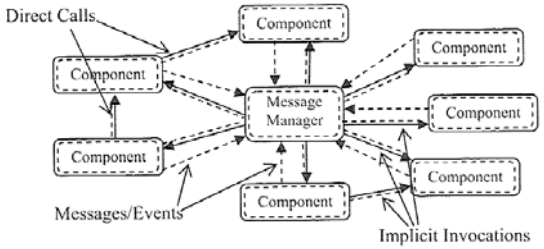
Disadvantages

- Need to know names of communicating processes/machines
- Communication across a network may be slow

16

Event-based Implicit Invocation Systems

- Components:
  - A number of independent processes or objects
- Connectors:
  - Communicate through messages



17

Purpose

The Event Systems architecture supports system modifiability, scalability, and performance.

- A component announces (broadcasts) one or more events.
- System Components register interest in an event by associating a procedure with it.
- The system invokes all events which have registered with it.
- Event announcement ``implicitly'' causes the invocation of procedures in other models.
- This style originates in constraint satisfaction (Planning), daemons, and packet-switched networks.
  - Used in Planning Domains
- Architectural components are modules whose interface provides both a collection of procedures and a set of events.
- Procedures may be called normally or be registered with events in the system.
- Implicit invocation systems are used in:
  - Programming environments to integrate tools
  - Database management systems to ensure consistency constraints
  - User interfaces to separate data from representation

Motivation

The Event Systems pattern consists of a number of independent processes or objects that communicate through messages. They send data to each other but typically do not directly control each other. Generally, the messages are passed among unnamed participants.

Application

The Event Systems pattern embodies control as part of the model. Individual components announce data that they wish to share (publish) with their environment. Other components may register an interest in this class of data (subscribe). If they do so, when the data appears, they are invoked and receive the data. Typical examples of this architecture are graphical user interfaces, which respond to mouse and keyboard events.

Event-based Implicit Invocation Systems

Advantages	Disadvantages
■ Component implementation does not need to know the name of its subscribers. ■ Components can run in parallel ■ Control is decoupled from individual components ■ Can process real-time events quickly	■ Implementation is more complex ■ Harder to test thoroughly ■ Subsystems don't know if or when events will be handled.

Advantages:

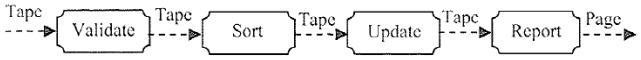
- Allows any component to register for events
- Eases system evolution by allowing components to be replaced without affecting the interfaces of other components in the system.

Disadvantages:

- Components relinquish control over the computation performed by the system.
- A component cannot assume that other components will respond to its requests
- A component does not know in which order events will be processed.
- In systems with a shared repository of data the performance and accuracy of the resource manager can become critical.
- Reasoning about correctness can be difficult because the meaning of a procedure that announces events will depend on the context in which it was invoked.

Batch Sequential Processing

- The Batch Sequential pattern supports a system composed of reusable components, is easily modified, and may increase performance.
- Components:
  - Called processing steps
  - Independent programs
- Connectors:
  - Data and control passing from step to step



Purpose

The Batch Sequential pattern supports a system composed of reusable components, is easily modified, and may increase performance. In the batch sequential style, components are independent programs. They are executable, i.e. one component runs to completion before the next starts. The data is transmitted between components as a whole batch rather than a stream of data elements

Motivation

The Batch Sequential pattern is characterized by viewing the system as a series of transformations on successive pieces of input data. Data enters the system and flows through the components one at a time until they are assigned to some final destination (standard output or a data store). In the Batch Sequential pattern, each processing step (component) is an independent program. The assumption is that each step runs to completion before the next step starts. Each batch of data is transmitted as a whole between the steps.

Application

Each component maintains its own communication and control. Generally, each component is an individual program. The typical application for the Batch Sequential pattern is classical data processing. Another example may be a set of XSL transforms run in series.

Batch Sequential Processing

Advantages

- Quickly process large amounts of data
- Interchangeable components
- Jobs are processed in the background, no user monitoring

Disadvantages

- Generally expensive hardware
- Interactive applications are difficult to create

Advantages

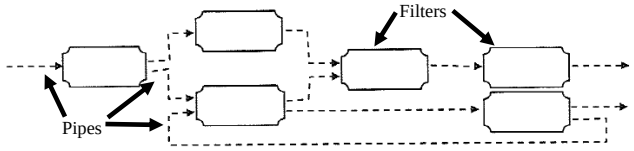
- Allows the designer to understand the system in terms of business process steps.
- Easy to maintain (supposedly) and add new or replace programs. However experience has shown that program and data stores are really tied to the business process.

Disadvantages

- Not good at interactive applications
- Limited Support for concurrent execution as each program needs all the data before it starts.
- Need to get all the data through the system before we can really see results.
- Not responsive to changes, no event handling, no fault tolerance, many problems if tapes are run out of sequence.

Pipe-and-Filter

- The Pipe-and-Filter architecture supports component reuse and application modifiability.
- Component: Filters
  - Read streams of data on its inputs and produces streams of data on its outputs
  - Apply a local transformation to the input streams and compute incrementally
- Connector: Pipes
  - Conduits for the streams, transmitting outputs of one filter to inputs of another



Purpose

The Pipe-and-Filter architecture supports component reuse and application modifiability.

Motivation

The Pipe-and-Filter pattern is characterized by viewing the system as a series of transformations on successive pieces of input data. Data enters the system and flows through the components one at a time until they are assigned to some final destination. Filters are stream transducers that incrementally transform data, use little contextual information, and retain no state information between instantiations. Pipes are stateless and simply exist to move data between filters.

Application

Both pipes and filters run until no more computations or transmissions are possible. Constraints on the pipe-and-filter pattern indicate the ways in which the pipes and filters can be joined. A pipe has a source end that can only be connected to a filter's output port and a sink end that can only be connected to a filter's input port. Any combination of filters connected by pipes can be packaged and appear to the external world as a filter. A common example of this architecture is the way in which various filters can be piped together in UNIX.

Pipe-and-Filter

- Constraints:
  - Filters must be independent entities
  - Filters do not know the identity of their upstream and downstream filters
  - The correctness of the system should not depend on the order in which the filters perform their incremental processing
- Example:
  - ls -l | more (command in Unix)

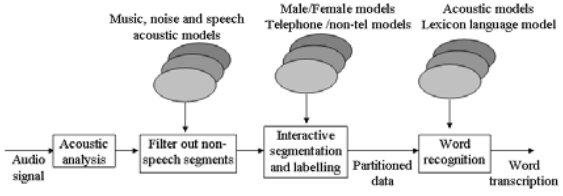
Constraints

There are three important constraints on the components and the connectors of Pipe-and-Filter architecture

- **Independence** - Filters must be independent entities, they should not share state with each others. There should be no global state variables as well.
- **Anonymity** - 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.
- **Concurrency** - 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. The dynamic execution can be best understood by considering all the filters are executing in parallel or concurrently.

Example

- Unix system '***ls - l | more***' combines two programs '***ls***' and '***more***' through the pipe connector '|'
- The system automatically generates text transcriptions from audio data streams of English speech in radio broadcast.



Pipe-and-Filter

Advantages

- No complex component interactions to manage
- Easily made parallel or distributed

Disadvantages

- Interactive applications are difficult to create
- Performance is frequently poor

Advantages

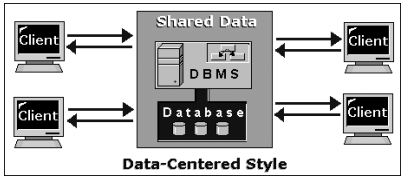
- Easy to understand
- Allow the designer to understand the system in terms of composition of filters.
- Support reuse
- Easy to maintain and enhance:
  - New filters can be added to existing systems
  - Old filters can be replaced by improved ones
- Permit specialized analyses (Throughput & deadlock)
- Naturally support concurrent execution
- Each filter can be implemented as a separate task and executed in parallel with other filters.

Disadvantages

- Not good at interactive applications, incremental display updates
- May need to maintain connections between separate yet related streams.
  - Different filters types may therefore require a common representation (packing & unpacking costs)
  - Each event handled from font to back.
- May force a lowest common denominator on data transmission
- Hampered by having to maintain correspondence between two separate but related streams

Data Repository

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



Purpose

The Data Repository pattern supports a system that is scalable, modifiable, and whose purpose is focused primarily on large amounts of data.

Motivation

The Data Repository pattern is useful for systems in which data access and update is shared by a number of individual clients. This pattern isolates the data store functionality from the data manipulation functionality. When the clients are built as independently executing processes, this pattern evolves into the client-server pattern.

Application

A client runs on an independent thread of control. The shared data store is a passive repository. Typical implementations of this pattern are database applications.

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.

Data Repository

Advantages

- Clients are relatively independent of each other.
- The data store is independent of the clients.
- New clients can be easily added.
- Provides an efficient way to share large amounts of data.
- No need to transmit data explicitly from one subsystem to another.

Disadvantages

- Communication between clients may be slow.
- Subsystems must agree on the database structure.

Components:

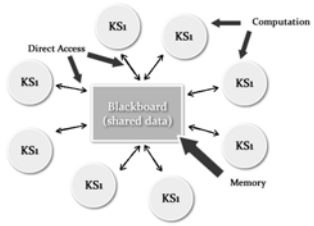
- Widely accessible data stores
  - Passive Repository such as File
  - Active Repository such as a Blackboard
- Clients - computation components

Connectors:

- Repository: independent thread of control
- Blackboard: activated by the blackboard

Blackboard

- Characteristics: cooperating 'partial solution solvers' collaborating but not following a pre-defined strategy.
  - Current state of the solution stored in the blackboard.
  - Processing triggered by the state of the blackboard.



Purpose

The Blackboard pattern supports a system that is scalable, modifiable, and whose purpose is focused primarily on large amounts of data.

Motivation

Similar to the Data Repository, the Blackboard pattern is useful for systems in which data access and update is shared by a number of individual clients. The primary difference from the Data Repository pattern is that the Blackboard sends notification to

subscribers when data of interest changes. This pattern isolates the data store functionality from the data manipulation functionality. When the clients are built as independently executing processes, this pattern evolves into the client-server pattern.

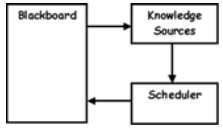
Application

A client runs on an independent thread of control. The shared data store is an active repository. In an active repository, clients subscribe to receive notification when data is updated. The repository then publishes notification to all subscribers when data of interest is updated.

Architecture

A blackboard model usually has three components:

- ***The Knowledge Source:*** independent pieces of application specific knowledge. Interaction between knowledge sources takes place only through the blackboard.
- ***The Blackboard Data Structure:*** 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 by the state of the blackboard. Knowledge sources respond opportunistically when changes in the blackboard make them applicable.



Blackboard

Advantages

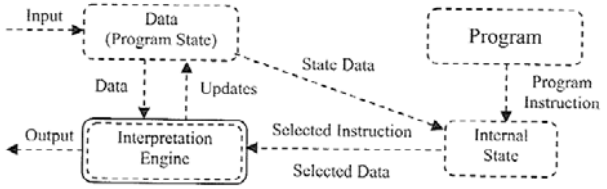
- Clients are relatively independent of each other.
- The data store is independent of the clients.
- New clients can be easily added.

Disadvantages

- Communication between clients may be slow.

Interpreter

- Architecture is based on a virtual machine produced in software.
- Special kind of a layered architecture where a layer is implemented as a true language interpreter.
- Components are 'program' being executed, its data, the interpretation engine and its state.
- Example: Java Virtual Machine.



Purpose

The Interpreter architecture supports system portability.

Motivation

The Interpreter pattern allows a program to run on a machine without compiling the program into native code. This is typical of virtual machines in general, which simulate some functionality that is not native to the hardware on which it is implemented.

Application

An interpreter allows a program to be built on a machine that simulates the actual production machine. It can also simulate disaster modes that would be too complex, costly, or dangerous to test with a real system. Popular examples of this architecture are the Java Virtual Machine (Java) and the Common Language Runtime (MS .NET). This architecture allows the language to be platform independent.

Components

Command interpreter, program/interpreter state, user interface.

Connectors

Typically very closely bound with direct procedure calls and shared state.

Example

Java Virtual Machine. Java code translated to platform independent bytecodes. JVM is platform specific and interprets (or compiles - JIT) the bytecodes.

Interpreter

Advantages

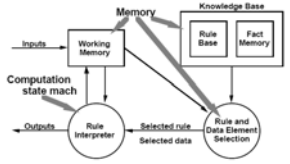
- Able to interrupt a program at run time
- Able to query a program at run time
- Able to modify a program at run time

Disadvantages

- Performance may be lower than native code.

Rule-Based System

- 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.



Purpose

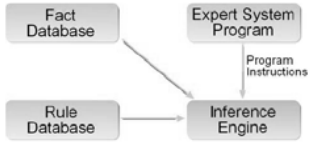
The Rule-Based System architecture supports system portability.

Motivation

The Rule-Based System pattern is used to simulate a human decision making process. This is typical of virtual machines in general, which simulate systems that would be too complex to build as a real system.

Application

A Rule-Based System allows a program to simulate the human decision making process. After a thorough analysis of the systems requirements, a set of facts will be collected and placed into the Fact Database. The relationships between these facts are stored in the Rule Database. The Inference Engine uses these two databases in combination with the user data to infer a solution to a given input. An example of this architecture is an expert system, which given a set of characteristics, can be used to identify an animal's genus in a biological database.



Rule-Based System

Advantages

- Able to interrupt a program at run time
- Able to query a program at run time
- Able to modify a program at run time

Disadvantages

- Performance may be lower than native code.

Case Study

- Keyword Frequency Vector (KFV)
- Keyword in Context (KWIC)

Case Study 1: KFV

- **The Problem:** Keyword Frequency Vector
  - The Keyword Frequency Vector (KFV) of a text file is a sequence of pairs of keywords and their frequency of appearance in the text
    - A good representation of the contents of a text
    - Widely used in information retrieval
    - Can be extracted from texts automatically
    - Small words (such as ‘a’, ‘the’, ‘is’, ‘it’) are removed from the vector
    - The same word of different forms should be treated as one

Example of KFV

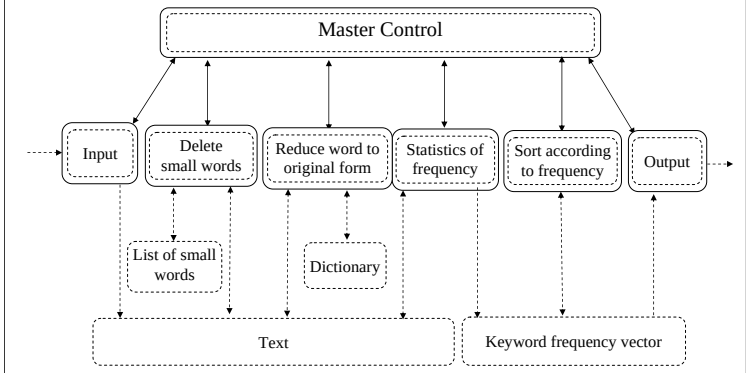
- **Input**

The keyword frequency vector of a text file is a sequence of pairs of keywords and their frequency of appearance in the text.
It is a good representation of the contents of the text.
Keyword frequency vectors are widely used in information retrieval.
For example the following is the keyword frequency vector of this paragraph.

- **Output**

Word	Frequency
keyword	4
frequency	4
text	4
vector	4
appearance	1
content	1
example	1
file	1
follow	1
good	1
information	1
pair	1
paragraph	1
representation	1
retrieval	1
sequence	1
use	1
widely	1

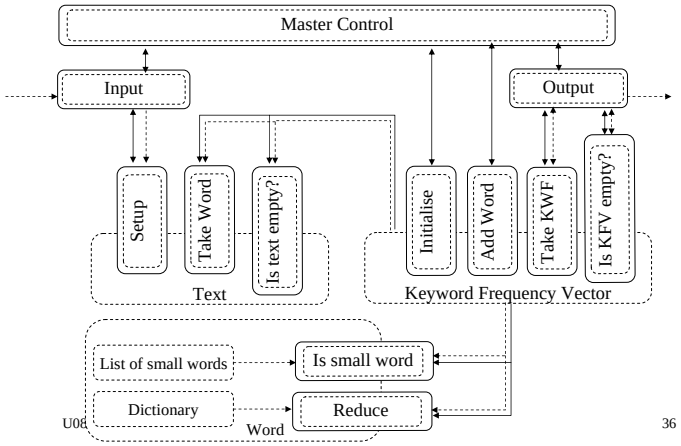
Design 1: Main Program/Subroutines with Shared Data



There are 6 major components:

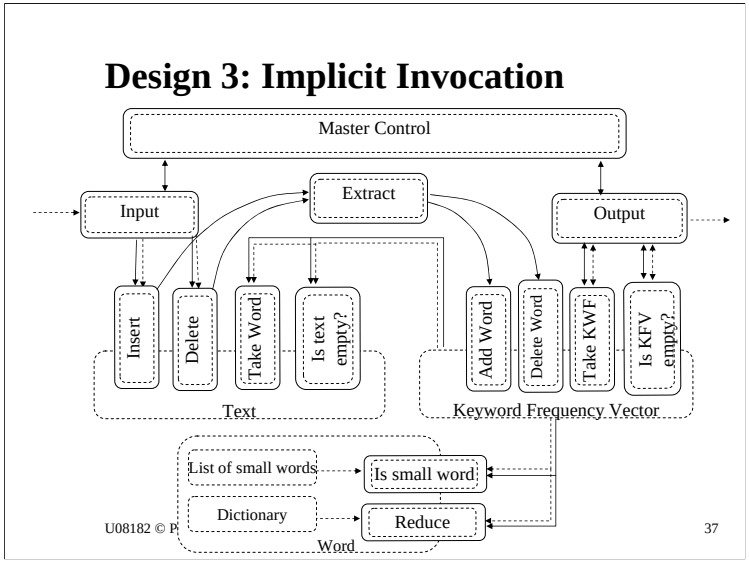
1. Input
   - To get the input text from input device or any other source of information
   - To store the text into internal memory in an appropriate format
   - The design of internal format will be determined by detailed design
2. Delete small words
   - The small words contained in the text are deleted from the text as it is stored in the internal memory
   - It will use a list of small words
3. Reduce word to its original form
   - Each word left in the text are then reduced to its original form
     - 'Architectures' ➔ 'architecture'
     - 'Calculi' ➔'calculus'
     - 'Followed' ➔'follow'
   - A dictionary will be used
4. Statistics of frequency
   - To count the occurrences of a word in the text to generate a sequence of pairs comprising the word and its frequency
   - This sequence of pairs is not necessarily ordered according to the frequency, but may be in the alphabetic order of keywords
   - The result will be stored in another memory storage
5. Sort according to the frequency
   - Sort the sequence of pairs of keywords and their frequencies into an order according to the frequency
6. Output
   - Translate the keyword frequency vector into required output format
   - Output to the device

Design 2: Abstract Data Type



There are three Abstract Data Type in this design:

1. Word ADT
   - Is-small-word:
     - A Boolean function that checks if the parameter is a small word.
     - It returns TRUE if the word is listed in a list of small words, otherwise it returns FALSE.
   - Reduce:
     - A function on words
     - It changes a word to its original form according to a dictionary of words and returns back.
2. Text ADT
   - Setup:
     - get the text from the input component
     - translate the text into an internal format
     - stores the text in its internal data storage
   - Take-word:
     - a function that returns one word in the text and deletes it from its internal data storage.
   - Is-text-empty:
     - A Boolean function that returns TRUE if the internal storage is empty, otherwise returns FALSE when it contains at least one word.
3. Keyword Frequency Vector ADT
   - Initialise:
     - It initialises the internal representation of the vector
   - Add-word: adds a word to the keyword frequency vector
     - Calls is-small-word function of the word ADT
     - If the function returns TRUE, then do nothing
     - ELSE calls the reduce function of the word ADT and searches the keyword frequency vector
     - If the vector already contains the word, then its frequency is added by 1, else the keyword is added into the vector with frequency 1
   - Take-KWF:
     - Return the frequency and keyword of highest frequency
     - Delete the keyword from the vector
   - Is-KFV-empty:
     - A Boolean function that returns TRUE if the vector is empty, otherwise, it returns FALSE



•The implicit invocation architecture also use three abstract data types to access the data abstractly.

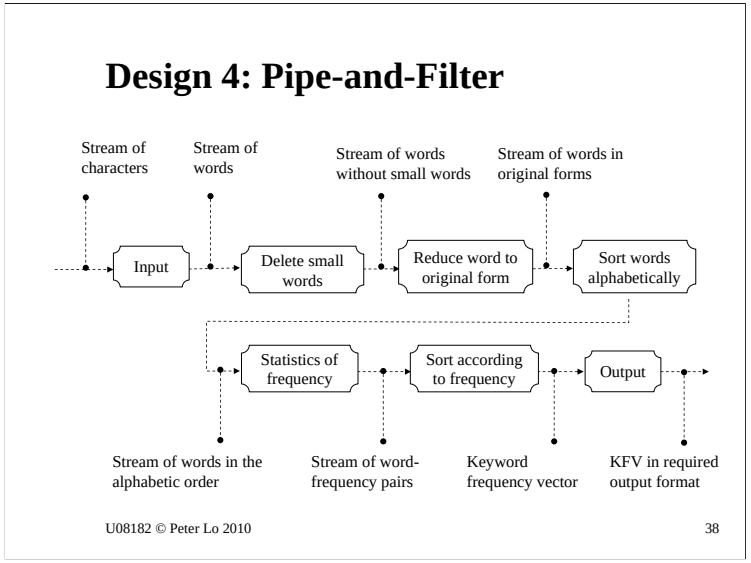
•The computations are invoked implicitly when data is modified.

•Each time when the data is modified, an event is generated and the event drives a corresponding event handling function to execute.

•Interactions are based on an active data model.

•The act of inserting or deleting a word from the text will cause the extract component to call the add-word or delete-word operation on the keyword frequency vector, which consequently change the vector's value.

•This allows the system to produce keyword frequency vector interactively and keep the stored vector consistent with the text while the user is editing the text.



There are 7 major components:

1. Input
   - Takes the stream of characters and breaks it down to a stream of words.
2. Delete small words
   - Removes the small words in the input stream of words
3. Reduce words to original forms
   - Changes each word in the stream of words into their original forms
4. Sort words alphabetically
   - Takes the stream of words and sort it into alphabetical order
5. Count the frequency
   - Count the occurrences of each word in the stream and generates a stream of keyword-frequency pairs
6. Sort vector according to frequency
   - Sort the stream of keyword-frequency pairs according to frequency
7. Output
   - Takes a stream of keyword-frequency pairs that is sorted according to the frequency and generates a keyword frequency vector in the required output format

The Quality Concerns

- Quality Attributes to be Considered:
  - Modifiability
    - Modifiability with regard to changes in the processing algorithm
    - Modifiability with regard to changes in data representation
    - Modifiability with regard to enhancement to system function
  - Performance
  - Reusability

Modifiability with regard to changes in the processing algorithm:

- To extract KFV incrementally paragraph by paragraph as it is read from the input device
- To extract KFV on the whole text file after they are read
- To extract on demand when the KFV is required

Modifiability with regard to changes in data representation:

- To store text, words and characters in various ways
- To store the KFV explicitly or implicitly

Modifiability with regard to enhancement to system function:

- To treat synonyms as the same word
- To change the systems to be interactive, and allow the user to delete and insert words from the original text

Performance:

- The performance of the system in terms of space used and the time needed to execute the program

Reusability:

- To what extend can the components are reusable

Comparison of Architectures for KFV

Attribute Architecture	Shared data	Abstract data type	Implicit invocation	Pipe-filter
Change in algorithm	–	–	+	+
Change in data representation	–	+	+	+
Change in function	+	–	+	–
Performance	+	+	–	–
Reuse	–	+	+	+

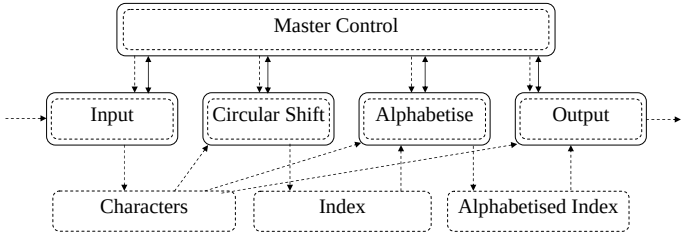
Case Study 2: KWIC

- **The Problem:** Keyword in Context
  - Input:
    - An ordered sequence of lines of text.
    - Each line is an ordered sequence of words
    - Each word is an ordered sequence of characters
  - Output:
    - Lines are ‘circularly shifted’ by repeatedly removing the first word and appending it at the end of the line.
    - Outputs a listing of all circular shifts of all lines in alphabetical order.

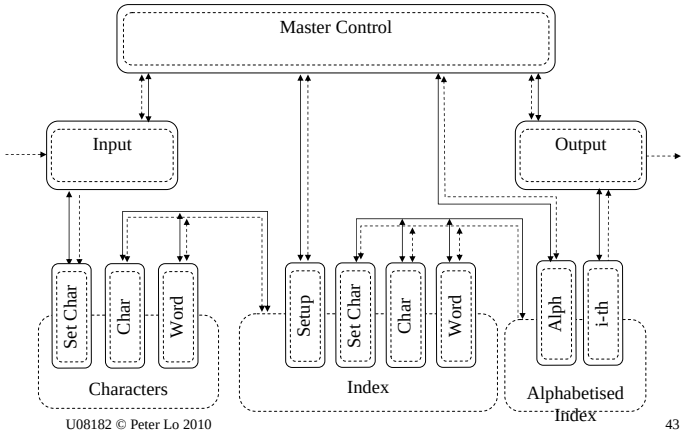
Example of KWIC

- Input: sequence of lines
  - Key word in context
- Output: circularly shifted, alphabetically ordered lines
  - Context key word in
  - In context key word
  - Key word in context
  - Word in context key

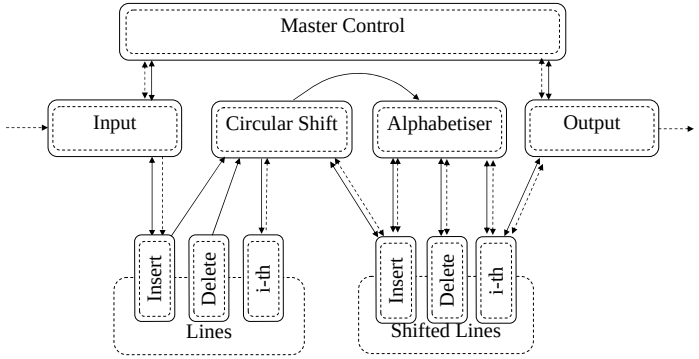
Design 1: Main Program/Subroutines with Shared Data



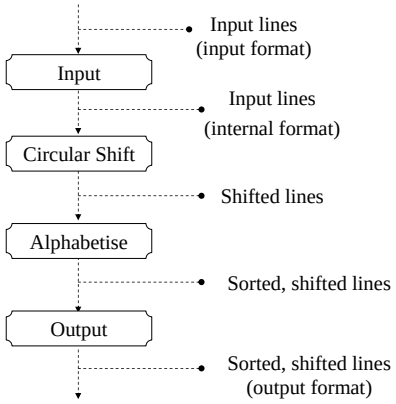
Design 2: Abstract Data Types



Design 3: Implicit Invocation



Design 4: Pipe-and-Filter



The Quality Concerns

- Quality Attributes to be Considered
  - Modifiability
    - Modifiability with regard to changes in the processing algorithm
    - Modifiability with regard to changes in data representation
    - Modifiability with regard to enhancement to system function
  - Performance
  - Reusability

Quality Attributes to be Considered

- Modifiability
  - Modifiability with regard to changes in the processing algorithm
    - To process on each line as it is read from the input device
    - To process on all the lines after they are read
    - To process on demand when the alphabetisation requires a new set of shifted lines
  - Modifiability with regard to changes in data representation
    - To store lines, words, and characters in various ways
    - To represent circular shifted lines explicitly or implicitly (as pairs of index and offset)
  - Modifiability with regard to enhancement to system function
    - To eliminate circular shifts that start with certain noise words (such as a, an, and, etc.)
    - To be interactive, and allow the user to delete lines from the original lists (or from the circular shifted lists)
- Performance
  - The performance of the system in terms of space used and the time needed to execute the program
- Reusability
  - To what extend can the components serve as reusable entities?

Comparisons of Architectures for KWIC

Attribute Architecture	Shared data	Abstract data type	Implicit invocation	Pipe-and-filter
Change in Algorithm	–	–	+	+
Change in Data Representation	–	+	–	–
Change in Function	+	–	+	+
Performance	+	+	–	–
Reuse	–	+	+	+

Concluding Remarks

•There is no architectural design that is satisfactory over all design considerations.

- •A trade-off must be made in on selection of a design.
- •This decision must be based on good understanding of the priority of the requirements.

•There is a pattern of quality attributes for an architectural style used in different problems.

- •Architectural styles can provide a useful guidance of design to achieve quality attributes.

•Two tables are not identical.

- •Some quality attributes are not uniquely determined by the architectural style.
- •It is necessary to analyse an architectural design against the details of quality requirements.

Further Readings

- Zhu, H., Software Design methodology. Chapter 7, pp172-198.
- Shaw, M and Garlan, D., Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996. Chapter 3: Case studies, pp33~68.
- Bass, L., Clements, P. and Kazman, R., Software Architecture in Practice, Addison Wesley, 1998.
  - Chapter 5: Moving From Qualities to Architecture: Architectural Styles, pp93~122
  - Chapter 7; The World Wide Web, pp145~163
  - Chapter 8: CORBA, pp165~187

A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems

http://academic.research.microsoft.com/Paper/341519.aspx