FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

UNIT 4
ARCHITECTURAL PATTERNS-1

Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them.

2.1 INTRODUCTION

Architectural patterns represent the highest-level patterns in our pattern system. They help you to satisfy the fundamental structure of an application. We group our patterns into four categories:

- ***From Mud to Structure.*** The category includes the Layers pattern, the Pipes and Filters pattern and the Blackboard pattern.
- ***Distributed systems.*** This category includes one pattern, Broker and refers to two patterns in other categories, Microkernel and Pipes and Filters.
- ***Interactive systems.*** This category comprises two patterns, the Model-View-Controller pattern and the Presentation-Abstraction-Control pattern.
- ***Adaptable systems.*** The Reflection pattern and the Microkernel pattern strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

2.2 FROM MUD TO STRUCTURE

Before we start the design of a new system, we collect the requirements from the customer and transform them into specifications. The next major technical task is to define the architecture of the system. This means finding a high level subdivision of the system into constitute parts.
We describe three architectural patterns that provide high level system subdivisions of different kinds:

- The ***Layers*** pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.
- The ***Pipes and Filters*** pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.
- The ***Blackboard*** pattern is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

[If mud to structure question is asked for 10 marks in exam, summary of full unit about layers, pipes & filters and blackboard pattern should be written]

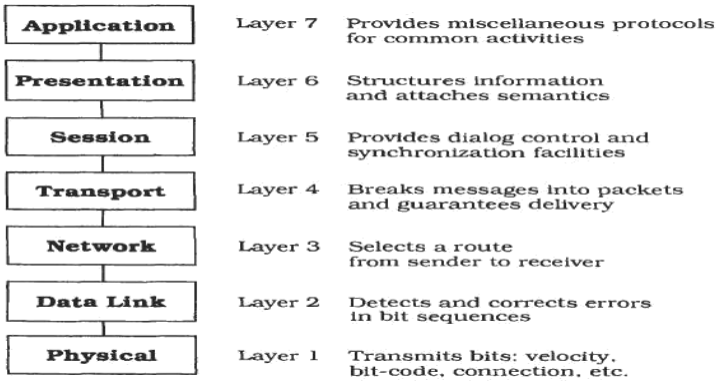
LAYERS

The layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

Example: Networking protocols are best example of layered architectures. Such a protocol consists of a set of rules and conventions that describes how computer programmer communicates across machine boundaries. The format, contacts and meaning of all messages are defined. The protocol specifies agreements at a variety of abstraction levels, ranging from the details of bit transmission to high level abstraction logic.
Therefore the designers use secured sub protocols and arrange them in layers. Each layer deals with a specific aspect of communication and users the services of the next lower layer. (see diagram & explain more)

Context: a large system that requires decomposition.

***Problem*: THE SYSTEM WE ARE BUILDING IS DIVIDED BY MIX OF LOW AND HIGH LEVEL ISSUES, WHERE HIGH-LEVEL OPERATIONS RELY ON THE LOWER-LEVEL ONES. FOR EX, HIGH-LEVEL WILL BE INTERACTIVE TO USER AND LOW-LEVEL WILL BE CONCERNED WITH HARDWARE IMPLEMENTATION.**



In such a case, we need to balance the following forces:

- Late source code changes should not ripple through the systems. They should be confined to one component and not affect others.
- Interfaces should be stable, and may even be impressed by a standards body.
- Parts of the system should be exchangeable (i.e, a particular layer can be changed).
- It may be necessary to build other systems at a later date with the same low-level issues as the system you are currently designing.
- Similar responsibilities should be grouped to help understandability and maintainability.
- There is no 'standard' component granularity.
- Complex components need further decomposition.
- Crossing component boundaries may impede performance, for example when a substantial amount of data must be transferred over several boundaries.
- The system will be built by a team of programmers, and works has to be subdivided along clear boundaries.

Solution:

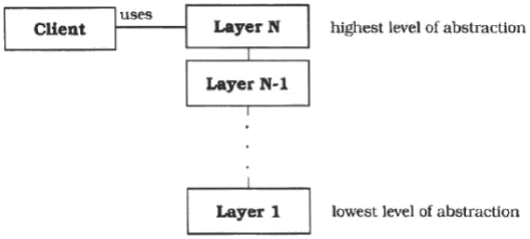
- Structure your system into an appropriate number of layers and place them on top of each other.
- Lowest layer is called 1 (base of our system), the highest is called layer N. i.e, Layer 1, ……. Layer J-1, Layer J, ….. Layer N.
- Most of the services that layer J Provides are composed of services provided by layer J-1. In other words, the services of each layer implement a strategy for combining the services of the layer below in a meaningful way. In addition, layer J's services may depend on other services in layer J.

Structure:

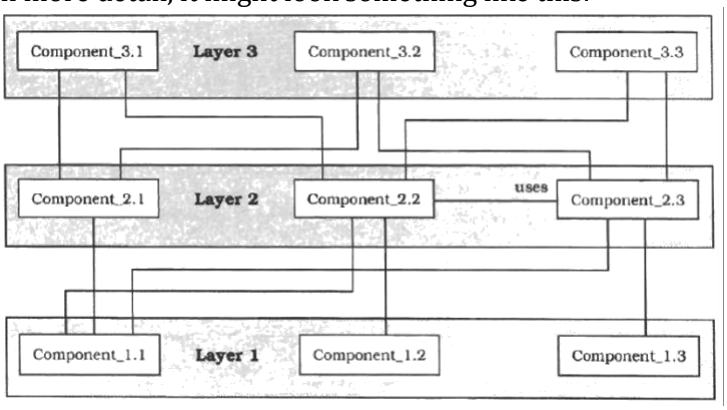
- An individual layer can be described by the following CRC card:

Class Layer J	*Collaborator* • Layer J-1
Responsibility • Provides services used by Layer J+1. • Delegates subtasks to Layer J-1.	

- The main structural characteristics of the layers patterns is that services of layer J are only use by layer J+1-there are no further direct dependencies between layers. This structure can be compared with a stack, or even an onion. Each individual layer shields all lower from direct access by higher layers.

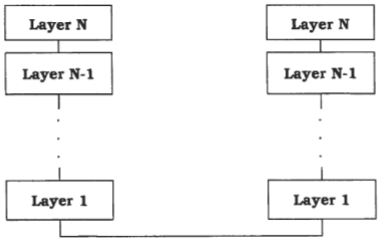


- In more detail, it might look something like this:



Dynamics:

- **Scenario I** is probably the best-known one. A client Issues a request to Layer N. Since Layer N cannot carry out the request on its own. It calls the next Layer N - 1 for supporting subtasks. Layer N - I provides these. In the process sending further requests to Layer N-2 and so on until Layer I is reached. Here, the lowest-level services are finally performed. If necessary, replies to the different requests are passed back up from Layer 1 to Layer 2, from Layer 2 to Layer 3, and so on until the final reply arrives at Layer N.
- **Scenario II** illustrates bottom-up communication-a chain of actions starts at Layer 1, for example when a device driver detects input. The driver translates the input into an internal format and reports it to Layer 2 which starts interpreting it, and so on. In this way data moves up through the layers until it arrives at the highest layer. While top-down information and control flow are often described as 'requests'. Bottom-up calls can be termed 'notifications'.
- **Scenario III** describes the situation where requests only travel through a subset of the layers. A top-level request may only go to the next lower level N- 1 if this level can satisfy the request. An example of this is where level N- 1 acts as a cache and a request from level N can be satisfied without being sent all the way down to Layer 1 and from here to a remote server.
- **Scenario IV** An event is detected in Layer 1, but stops at Layer **3** instead of travelling all the way up to Layer N. In a communication protocol, for example, a resend request may arrive from an impatient client who requested data some time ago. In the meantime the server has already sent the answer, and the answer and the re-send request cross. In this case, Layer **3** of the server side may notice this and intercept the re-send request without further action.
- **Scenario V** involves two stacks of N layers communicating with each other. This scenario is well-known from communication protocols where the stacks are known as 'protocol stacks'. In the following diagram, Layer N of the left stack issues a request. 'The request moves down through the layers until it reaches Layer **1,** is sent to Layer 1 of the right stack, and there moves up through the layers of the right stack. The response to the request follows the reverse path until it arrives at Layer N of the left stack.



Implementation:

The following steps describe a step-wise refinement approach to the definition of a layered architecture.

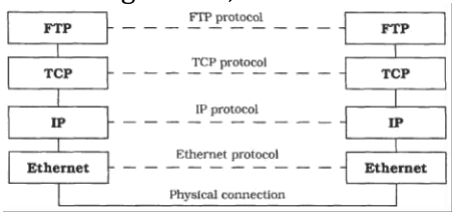
- **Define the abstraction criterion for grouping tasks into layers.**
  - This criterion is often the conceptual distance from the platform (sometimes, we encounter other abstraction paradigm as well).
  - In the real world of software development we often use a mix of abstraction criterions. For ex, the distance from the hardware can shape the lower levels, and conceptual complexity governs the higher ones.
  - Example layering obtained using mixed model layering principle is shown below User-visible elements
    - Specific application modules
    - Common services level
    - Operating system interface level
    - Operating system
    - Hardware
- **Determine the number of abstraction levels according to your abstraction criterion.** ○ Each abstraction level corresponds to one layer of the pattern.
  - Having too many layers may impose unnecessary overhead, while too few layers can result in a poor structure.
- **Name the layers and assign the tasks to each of them.**
  - The task of the highest layer is the overall system task, as perceived by the client.
  - The tasks of all other layers are to be helpers to higher layers.
- **Specify the services**
  - It is often better to locate more services in higher layers than in lower layers.
  - The base layers should be kept 'slim' while higher layers can expand to cover a spectrum of applicability.
  - This phenomenon is also called the '*inverted pyramid of reuse*'.
- **Refine the layering**
  - Iterate over steps 1 to 4.
  - It is not possible to define an abstraction criterion precisely before thinking about the implied layers and their services.
  - Alternatively it is wrong to define components and services first and later impose a layered structure.
  - The solution is to perform the first four steps several times until a natural and stable layering evolves.
- **Specify an interface for each layer.**
  - If layer J should be a 'black box' for layer J+1, design a flat interface that offers all layer J's services.
  - 'White box' approach is that in which, layer J+1 sees the internals of layer J.
  - 'Gray box' approach is a compromise between black and white box approaches. Here layer J+1 is aware of the fact that layer J consists of three components, and address them separately, but does not see the internal workings of individual components.
- **Structure individual layers**
  - When an individual layer is complex, it should be broken into separate components.
  - This subdivision can be helped by using finer-grained patterns.
- **Specify the communication between adjacent layers.**
  - Push model (most often used): when layer J invokes a service of layer J+1, any required information is passed as part of the service call.
  - Pull model: it is the reverse of the push model. It occurs when the lower layer fetches available information from the higher layer at its own discretion.
- **Decouple adjacent layers.**
  - For top-down communication, where requests travel top-down, we can use one-way coupling (i.e, upper layer is aware of the next lower layer, but the lower layer is unaware of the identity of its users) here return values are sufficient to transport the results in the reverse direction.

- For bottom-up communication, you can use callbacks and still preserve a top-down one way coupling. Here the upper layer registers callback functions with the lower layer.
- We can also decouple the upper layer from the lower layer to a certain degree using object oriented techniques.

★ **Design an error handling strategy**
- An error can either be handled in the layer where it occurred or be passed to the next higher layer.
- As a rule of thumb, try to handle the errors at the lowest layer possible.

Example resolved:

The most widely-used communication protocol, TCP/IP, does not strictly conform to the OSI model and consists of only four layers: TCP and IP constitute the middle layers, with the application at the top and the transport medium at the bottom. A typical configuration, that for the **UNIX** ftp's utility, is shown below:



Variants:

★ **Relaxed layered system:**
- Less restrictive about relationship between layers. Here each layer may use the services of all layers below it, not only of the next lower layer.
- A layer may also be partially opaque i.e. some of its services are only visible to the next higher layer, while others are visible to all higher layers.
- Advantage: flexibility and performance.
- Disadvantage: maintainability.

★ **Layering through inheritance**
- This variant can be found in some object oriented systems.
- Here, lower layers are implemented as base classes. A higher layer requesting services from a lower layer inherits from the lower layer's implementation and hence can issue requests to the base class services.
- Advantage: higher layers can modify lower-layer services according to their needs.
- Disadvantage: closely ties the higher layer to the lower layer.

Known uses:

★ **Virtual machines**: we can speak of lower levels as a virtual machine that insulates higher levels from low-level details or varying hardware. E.g. Java virtual machine.

★ **API'S**: An API is a layer that encapsulates lower layers of frequently-used functionality. An API is usually a flat collection Specifications, such as the UNIX system calls.

★ **Information system (IS):** IS from the business software domain often use a two-layer architecture. The bottom layer is a database that holds company-specific data. This architecture can be split into four-layer architecture. These are, from highest to lowest:
- ♠ Presentation
- ♠ Application logic
- ♠ Domain layer
- ♠ Database

★ **Windows NT:** This OS is structured according to the microkernel pattern. It has the following layers:
- ♠ System services
- ♠ Resource management layer
- ♠ Kernel
- ♠ HAL(hardware abstraction layer)
- ♠ Hardware

Consequences:

The layers pattern has several ***benefits***:

- **Reuse of layers**
  - If an individual layer embodies a well-defined abstraction and has a well-defined and documented interface, the layer can be reused in multiple contexts.
  - However, developers often prefer to rewrite its functionality in order to avoid higher costs of reusing existing layers.
- **Support for standardization**
  - Clearly-defined and commonly-accepted levels of abstraction enable the development of standardized tasks and interfaces.
  - Different implementations of the same interface can then be used interchangeably. This allows you to use products from different vendors in different layers.
- **Dependencies are kept local**
  - Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed.
- **Exchangeability**
  - Individual layer implementations can be replaced by semantically-equivalent implementations without too great an effort.

The layers pattern has several ***liabilities:***

- **Cascades of changing behavior**
  - A severe problem can occur when the behavior of the layer changes.
  - The layering becomes a disadvantage if you have to do a substantial amount of rework on many layers to incorporate an apparently local change.
- **Lower efficiency**
  - If high-level services in the upper layers rely heavily on the lowest layers, all relevant data must be transferred through a number of intermediate layers, and may be transformed several times. Therefore, layered architecture is less efficient than a monolithic structure or a 'sea of objects'.
- **Unnecessary work**
  - If some services performed by lower layers perform excessive or duplicate work not actually required by the higher layer this has a negative impact on performance.
  - De-multiplexing in a communication protocol stack is an example of this phenomenon.
- **Difficulty of establishing the correct granularity of layers**
  - The decision about the granularity of layers and the assignment of task to layers is difficult, but is critical for the architecture.

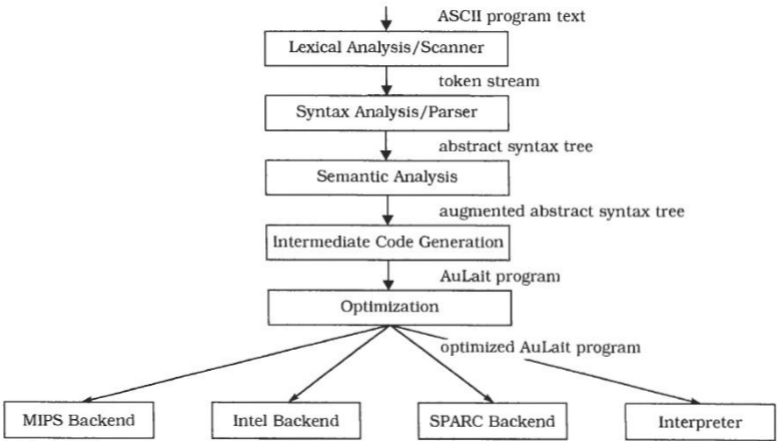
PIPES AND FILTERS

The pipes and filter's architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.

Example:

Suppose we have defined a new programming language called **Mocha** [*Modular Object Computation with Hypothetical Algorithms*]. Our task is to build a portable compiler for this language. To support existing and future hardware platforms we define an intermediate language **AuLait** [*Another Universal Language for Intermediate Translation*] running on a virtual machine Cup (Concurrent Uniform Processor).

Conceptually, translation from Mocha to AuLait consists of the phases lexical analysis, syntax analysis, semantic analysis, intermediate-code generation (AuLait), and optionally intermediate-code optimization. Each stage has well-defined input and output data.



Context:

Processing data streams.

Problem:

- Imagine you are building a system that must process or transform a stream of input data. Implementing such a system as a single component may not be feasible for several reasons: the system has to be built by several developers, the global system task decomposes naturally into several processing stages, and the requirements are likely to change.
- The design of the system-especially the interconnection of the processing steps-has to consider the following forces:
  - Future system enhancements should be possible by exchanging processing steps or by recombination of steps, even by users.
  - Small processing steps are easier to reuse in different contexts than larger contexts.
  - Non-adjacent processing steps do not share information.
  - Different sources of input data exists, such as a network connection or a hardware sensor providing temperature readings, for example.
  - It should be possible to present or store the final results in various ways.
  - Explicit storage of intermediate results for further processing the steps, for example running them in parallel or quasi-parallel.
  - You may not want to rule out multi-processing the steps

Solution:

- The pipes and filters architectural pattern divides the task of a system into several sequential processing steps (connected by the dataflow through the system).
- Each step is implemented by a filter component, which consumes and delivers data incrementally to achieve low battery and parallel processing.
- The input to a system is provided by a data source such as a text file.
- The output flows into a data sink such as a file, terminal, and so on.
- The data source, the filters, and the data sink are connected sequentially by pipes. Each pipe implements the data flow between adjacent processing steps.
- The sequence of filters combined by pipes is called a processing pipeline.

Structure:

- **Filter component:**
  - Filter components are the processing units of the pipeline.
  - A filter enriches, refines or transforms its input data. It enriches data by computing and adding information, refines data by concentrating or extracting information, and transforms data by delivering the data in some other representation.
  - It is responsible for the following activities:
    - The subsequent pipeline element pulls output data from the filter. (passive filter)

- The previous pipeline element pushes new input data to the filter. (passive filter)
- Most commonly, the filter is active in a loop, pulling its input from and pushing its output down the pipeline. (active filter)

Class Filter	Collaborators • Pipe
Responsibility • Gets input data. • Performs a function on its input data. • Supplies output data.	

★ **Pipe component:**

- Pipes denote the connection between filters, between the data source and the first filter, and between the last filter and the data sink.
- If two active components are joined, the pipe synchronizes them.
- This synchronization is done with a first-in- first-out buffer.

Class Pipe	Collaborators • Data Source • Data Sink • Filter
Responsibility • Transfers data. • Buffers data. • Synchronizes active neighbors.	

★ **Data source component:**

- The data source represents the input to the system, and provides a sequence of data values of the same structure or type.
- It is responsible for delivering input to the processing pipeline.

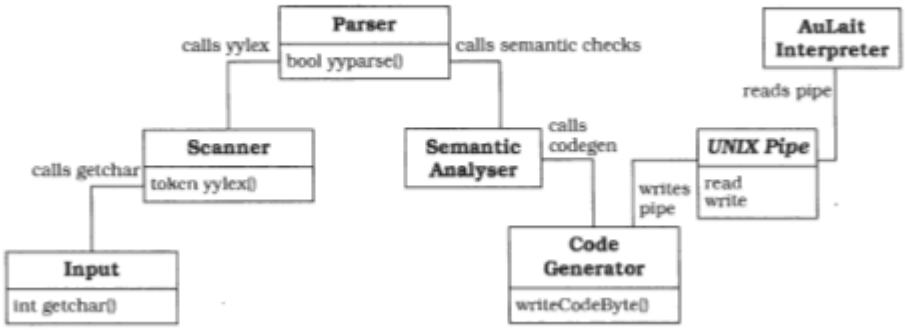
Class Data Source	Collaborators • Pipe
Responsibility • Delivers input to processing pipeline.	

★ **Data sink component:**

- The data sink collects the result from the end of the pipeline (i.e, it consumes output).
- Two variants of data sink:
  - An active data sink pulls results out of the preceding processing stage.
  - An passive data sink allows the preceding filter to push or write the results into it.

Class Data Sink	Collaborators • Pipe
Responsibility • Consumes output.	

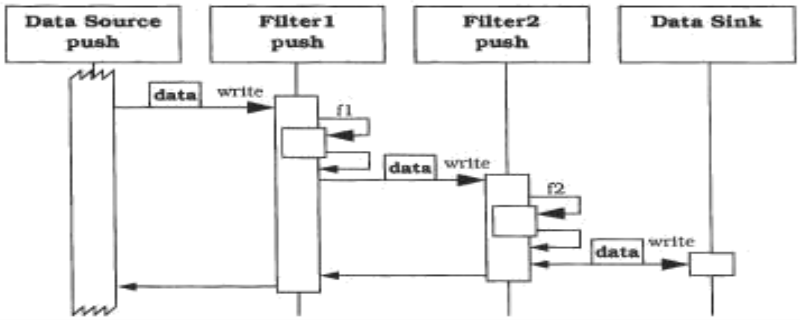
In our Mocha compiler we use the UNIX tools lex and yacc to implement the first two stages of the compiler. The connection to the other frontend stages consists of many procedure calls embedded in the grammar action rules, and not just simple data flow. The backends and interpreter run as separate programs to allow exchangeability. They are connected via a UNIX pipe to the frontend.



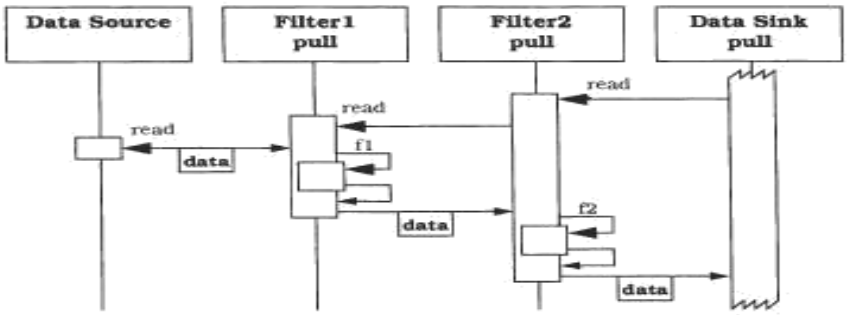
Dynamics:

The following scenarios show different options for control flow between adjacent filters.

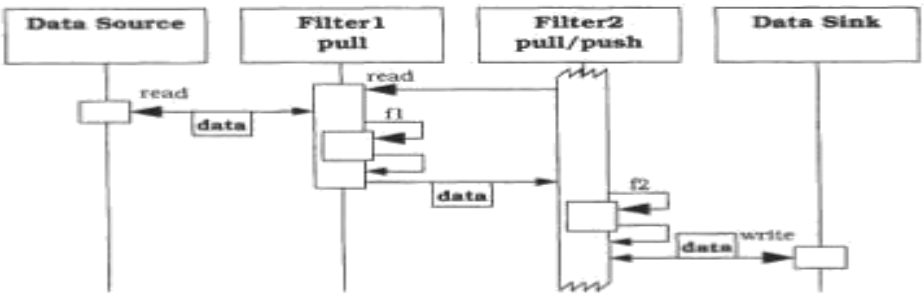
- **Scenario 1** shows a push pipeline in which activity starts with the data source.



- **Scenario 2** shows a pull pipeline in which control flow starts with the data sink.

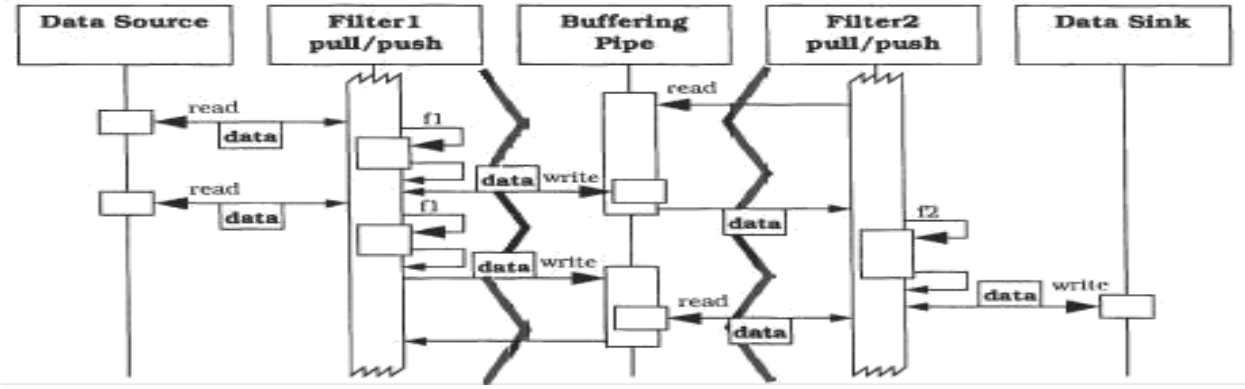


- **Scenario 3** shows a mixed push-pull pipeline with passive data source and sink. Here second filter plays the active role and starts the processing.



- **Scenario 4** shows a more complex but typical behavior of pipes and filter system. All filters actively pull, compute, and push data in a loop.
- The following steps occur in this scenario:
  - Filter 2 tries to get new data by reading from the pipe. Because no data is available the data request suspends the activity of Filter 2-the buffer is empty.
  - Filter 1 pulls data from the data source and performs function f 1.
  - Filter 1 then pushes the result to the pipe.
  - Filter 2 can now continue, because new input data is available.

- Filter l can also continue, because it is not blocked by a full buffer within the pipe.
- Filter 2 computes f 2 and writes its result to the data sink.
- In parallel with Filter 2's activity, Filter 1 computes the next result and tries to push it down the pipe. This call is blocked because Filter 2 is not waiting for data-the buffer is full.
- Filter 2 now reads new input data that is already available from the pipe. This releases Filter 1 so that it can now continue its processing.



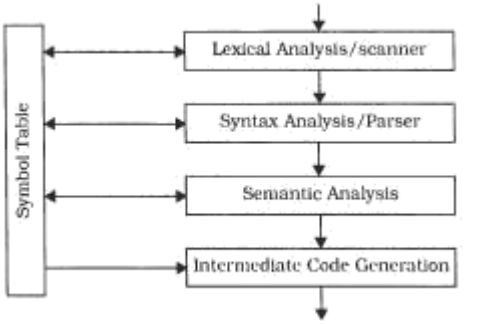
Implementation:

- **Divide the system's tasks into a sequence of processing stages.**
  - Each stage must depend only on the output of its direct predecessor.
  - All stages are conceptually connected by the data flow.
- **Define the data format to be passed along each pipe.**
  - Defining a uniform format results in the highest flexibility because it makes recombination of its filters easy.
  - You must also define how the end of input is marked.
- **Decide how to implement each pipe connection.**
  - This decision directly determines whether you implement the filters as active or passive components.
  - Adjacent pipes further define whether a passive filter is triggered by push or pull of data.
- **Design and implement the filters.**
  - Design of a filter component is based both on the task it must perform and on the adjacent pipes.
  - You can implement passive filters as a function, for pull activation, or as a procedure for push activation.
  - Active filters can be implemented either as processes or as threads in the pipeline program.
- **Design the error handling.**
  - Because the pipeline components do not share any global state, error handling is hard to address and is often neglected.
  - As a minimum, error detection should be possible. UNIX defines specific output channel for error messages, standard error.
  - If a filter detects error in its input data, it can ignore input until some clearly marked separation occurs.
  - It is hard to give a general strategy for error handling with a system based on the pipes and filter pattern.
- **Set up the processing pipeline.**
  - If your system handles a single task you can use a standardized main program that sets up the pipeline and starts processing.
  - You can increase the flexibility by providing a shell or other end-user facility to set up various pipelines from your set of filter components.

Example resolved:

We did not follow the Pipes and Filters pattern strictly in our Mocha compiler by implementing all phases of the compiler as separate filter programs connected by pipes. We combined the first four compiler phases into a

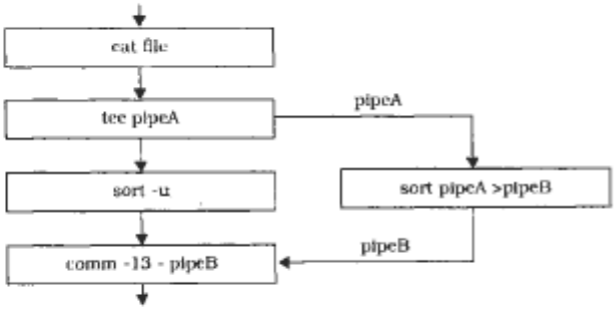
single program because they all access and modify the symbol table. This allows us to implement the pipe connection between the scanner and the parser as a simple function call.



Variants:

- Tee and join pipeline systems:
- The single-input single-output filter specification of the Pipes and Filters pattern can be varied to allow filters with more than one input and/or more than one output.
- For example, to build a sorted list of all lines that occur more than once In a text file. we can construct the following shell program:

```
# first create two auxiliary named pipes to be
used mknod pipeA p
mknod pipeB p
# now do the processing using available UNIX filters
# start side fork of processing in background:
sort pipeA > pipeB &
# the main pipeline
cat file | tee pipeA | sort -u | comm -13 - pipeB
```



Known uses:

★ **UNIX** [Bac86] popularized the Pipes and Filters paradigm. The flexibility of UNIX pipes made the operating system a suitable plafform for the binary reuse of filter programs and for application integration.

★ **CMS Pipelines** [HRV95] is an extension to the operating system of IBM mainframes to support Pipes and Filters architectures. It provides a reuse and integration platform in the same way as UNIX.

★ **LASSPTools** [Set95] is a toolset to support numerical analysis and graphics. The toolset consists mainly of filter programs that can be combined using UNIX pipes.

Consequences:

The Pipes and Filters architectural pattern has the following **benefits**

★ **No intermediate files necessary, but possible.**
- Computing results using separate programs is possible without pipes, by storing intermediate results in pipes.

★ **Flexibility by the filter change**
- Filters have simple interface that allows their easy exchange within a processing pipeline.

★ **Flexibility by recombination**
- This benefit combined with reusability of filter component allows you to create new processing pipelines by rearranging filters or by adding new ones.

- **Reuse of filter components.**
  - Support for recombination leads to easy reuse of filter components.
- **Rapid prototyping of pipelines.**
  - Easy to prototype a data processing system from existing filters.
- **Efficiency by parallel processing**.
  - It is possible to start active filter components in parallel in a multiprocessor system or a network.

The Pipes and Filters architectural pattern has the following **Liabilities**

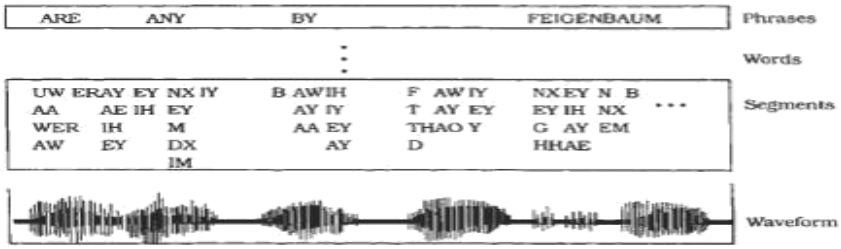
- **Sharing state information is expensive or inflexible**
  - This pattern is inefficient if your processing stage needs to share a large amount of global data.
- **Efficiency gain by parallel processing is often an illusion**, because:
  - The cost for transferring data between filters is relatively high compared to the cost of the computation carried out by a single filter.
  - Some filter consumes all their input before producing any output.
  - Context-switching between threads or processes is expensive on a single processor machine.
  - Synchronization of filters via pipes may start and stop filters often.
- **Data transformation overhead**
  - Using a single data type for all filter input and output to achieve highest flexibility results in data conversion overheads.
- **Error handling**
  - Is difficult. A concrete error-recovery or error-handling strategy depends on the task you need to solve.

BLACKBOARD

The blackboard architectural pattern is useful for problems for which no deterministic solution strategies are known. In blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

Example:

Consider a software system for speech recognition. The input to the system is speech recorded as a waveform. The system not only accepts single words, but also whole sentences that are restricted to the syntax and vocabulary needed for a specific application, such as a database query. The desired output is a machine representation of the corresponding English phrases.



Context:

An immediate domain in which no closed approach to a solution is known or feasible

Problem:

- A blackboard pattern tackle problems that do not have a feasible deterministic solution for the transformation of raw data into high level data structures, such as diagrams, tables, or English phrases.
- Examples of domains in which such problems occur are:- vision, image recognition, and speech recognition.
- They are characterized by problems that when decomposed into sub problems, spans several fields of expertise.
- The Solution to the partial problems requires different representations and paradigm.

- The following *forces* influence solutions to problems of this kind:
  - ✓ A complete search of the solution space is not feasible in a reasonable time.
  - ✓ Since the domain is immature, we may need to experiment with different algorithms for the same subtask. Hence, individual modules should be easily exchangeable.
  - ✓ There are different algorithms that solve partial problems.
  - ✓ Input as well as intermediate and final results, have different representation, and the algorithms are implemented according to different paradigms.
  - ✓ An algorithm usually works on the results of other algorithms.
  - ✓ Uncertain data and approximate solutions are involved.
  - ✓ Employing dis fact algorithms induces potential parallelism.

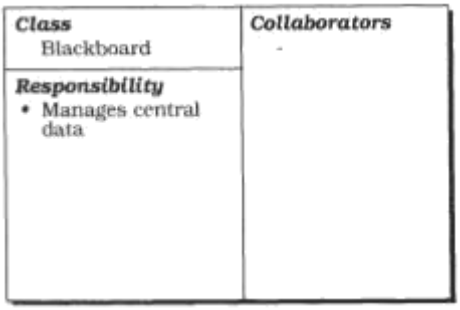
Solution:

- The idea behind the blackboard architecture is a collection of independent programs that work co operatively on a common data structure.
- Each program is meant for solving a particular part of overall task.
- These programs are independent of each other they do not call each other, nor there is a predefined sequence for their activation. Instead, the direction taken by the system is mainly determined by the current state of progress.
- A central control component evaluates the current state of processing and coordinates these programs.
- These data directed control regime is referred to as *opportunistic problem solving.*
- The set of all possible solutions is called *solution space* and is organized into levels of abstraction.
- The name 'blackboard' was chosen because it is reminiscent of the situation in which human experts sit in front of a real blackboard and work together to solve a problem.
- Each expert separately evaluates the current state of the solution, and may go up to the blackboard at any time and add, change or delete information.
- Humans usually decide themselves who has the next access to the blackboard.

Structure:

Divide your system into a component called a *blackboard*, a collection of *knowledge sources*, and a *control component.*

- **Blackboard:**
  - Blackboard is the central data store.
  - Elements of the solution space and control data are stored here.
  - Set of all data elements that can appear on the blackboard are referred to as vocabulary.
  - Blackboard provides an interface that enables all knowledge sources to read form and write to it.
  - We use the terms hypothesis or blackboard entry for solutions that are constructed during the problem solving process and put on the blackboard.
  - A hypothesis has several attributes, ex: its abstraction level.



- **Knowledge source**:
  - Knowledge sources are separate, independent subsystem that solve specific aspects of the overall problem.
  - Knowledge sources do not communicate directly they only read from and write to the blackboard.
  - Often a knowledge source operates on two levels of abstraction.
  - Each knowledge source is responsible for knowing the conditions under which it can contribute to a solution. Hence, knowledge sources are split into.

- *Condition part*: evaluates the current state of the solution process, as written on the blackboard, to determine if it can make a contribution.
- *Action part*: produces a result that may causes a change to the blackboard's content.

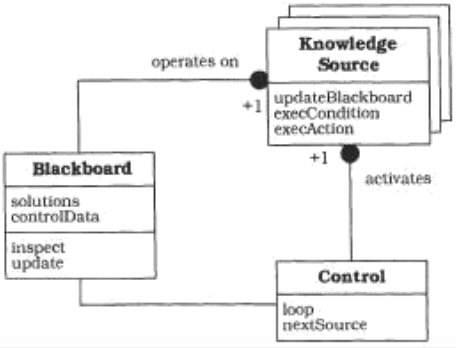
Class Knowledge Source	Collaborator • Blackboard
Responsibility • Evaluates its own applicability • Computes a result • Updates Black-board	

★ **Control component:**

- Runs a loop that monitors the changes on the blackboard and decides what action to take next.
- It schedules knowledge source evaluations and activations according to the knowledge applications strategy. The basis for this strategy is the data on the blackboard.

Class Control	Collaborators • Blackboard • Knowledge Source
Responsibility • Monitors Black-board • Schedules Know-ledge Source acti-vations	

- The following figure illustrates the relationship between the three components of the blackboard architecture.

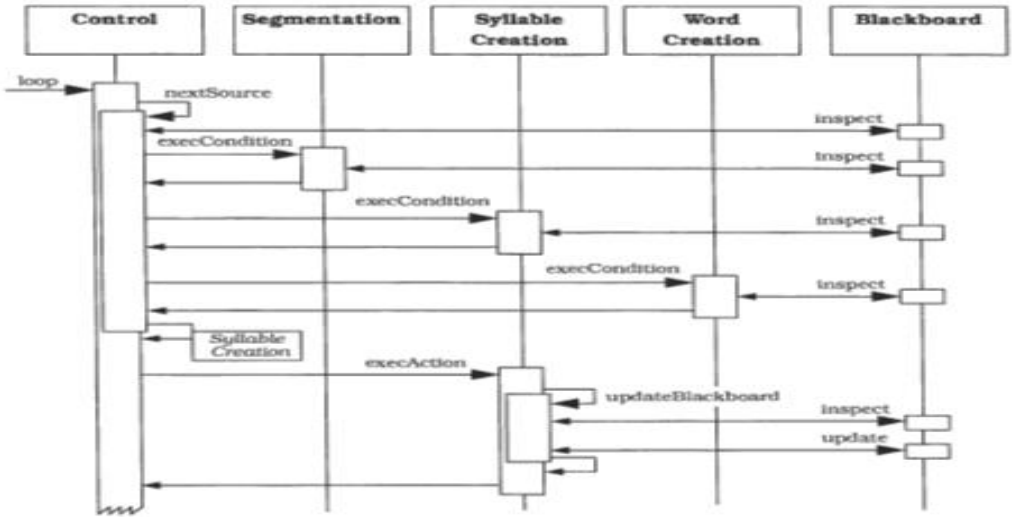


- Knowledge source calls `inspect()` to check the current solutions on the blackboard
- `Update()` is used to make changes to the data on the blackboard
- Control component runs a loop that monitors changes on the blackboard and decides what actions to take next. `nextSource()` is responsible for this decision.

Dynamics:

The following scenario illustrates the behavior of the Blackboard architecture. It is based on our speech recognition example:

- The main loop of the Control component is started.
- Control calls the **`nextsource`**`()` procedure to select the next knowledge source.
- **`nextsource`** `()` first determines which knowledge sources are potential contributors by observing the blackboard.
- **`nextsource()`** invokes the condition-part of each candidate knowledge source.
- The Control component chooses a knowledge source to invoke, and a hypothesis or a set of hypotheses to be worked on.



Implementation:

★ **Define the problem**
- ✓ Specify the domain of the problem
- ✓ Scrutinize the input to the system
- ✓ Define the o/p of the system
- ✓ Detail how the user interacts with the system.

★ **Define the solution space for the problem**
- ✓ Specify exactly what constitutes a top level solution
- ✓ List the different abstraction levels of solutions
- ✓ Organize solutions into one or more abstraction hierarchy.
- ✓ Find subdivisions of complete solutions that can be worked on independently.

★ **Divide the solution process into steps.**
- ✓ Define how solutions are transformed into higher level solutions.
- ✓ Describe how to predict hypothesis at the same abstraction levels.
- ✓ Detail how to verify predicted hypothesis by finding support for them in other levels.
- ✓ Specify the kind of knowledge that can be uses to exclude parts of the solution space.

★ **Divide the knowledge into specialized knowledge**
- ✓ These subtasks often correspond to areas of specialization.

★ **Define the vocabulary of the blackboard**
- ✓ Elaborate your first definition of the solution space and the abstraction levels of your solution.
- ✓ Find a representation for solutions that allows all knowledge sources to read from and contribute to the blackboard.
- ✓ The vocabulary of the blackboard cannot be defined of knowledge sources and the control component.

★ **Specify the control of the system.**
- ✓ Control component implements an opportunistic problem-solving strategy that determines which knowledge sources are allowed to make changes to the blackboard.
- ✓ The aim of this strategy is to construct a hypothesis that is acceptable as a result.
- ✓ The following mechanisms optimizes the evaluation of knowledge sources, and so increase the effectiveness and performance of control strategy.
- ✓ Classifying changes to the blackboard into two types. One type specify all blackboard change that may imply new set of applicable knowledge sources, and the other specifies all blackboard changes that do not.
- ✓ Associating categories of blackboard changes with sets of possibly applicable knowledge sources.
- ✓ Focusing of control. The focus contains either partial results on the blackboard or knowledge sources that should be preferred over others.
- ✓ Creating on queue in which knowledge sources classified as applicable wait for their execution.

★ **Implement the knowledge sources**

- ✓ Split the knowledge sources into condition parts and action-parts according to the needs of the control component.
- ✓ We can implement different knowledge sources in the same system using different technologies

Variants:

- ★ **Production systems:** used in oops language. Here the subroutines are represented as condition-action rules, and data is globally available in working.
- ★ **Repository:** it is a generalization of blackboard pattern the central data structure of this variant is called a repository.

Known uses:

- ★ **HEARSAY-II** → The first Blackboard system was the HEARSAY-II speech recognition system from the early 1970's. It was developed as a natural language interface to a literature database. Its task was to answer queries about documents and to retrieve documents from a collection of abstracts of Artificial Intelligence publications. The inputs to the system were acoustic signals that were semantically interpreted and then transformed to a database query. The control component of HEARSAY-I1 consists of the following:
  - The focus of control database, which contains a table of primitive change types of blackboard changes, and those condition-parts that can be executed for each change type.
  - The scheduling queue, which contains pointers to condition- or action-parts of knowledge source.
  - The monitor, which keeps track of each change made to the blackboard.
  - The scheduler, which uses experimentally-derived heuristics to calculate priorities for the condition- and action- parts waiting in the scheduling queue.
- ★ HASP/SIAP
- ★ CRYSALIS
- ★ TRICERO
- ★ SUS

[Refer text if you need more details]

Example resolved:

- ★ RPOL – runs as a high-priority task & calculates overall rating for the new hypothesis
- ★ PREDICT – works on a phrase and generates predictions of all words
- ★ VERIFY – verify the existence of, or reject, a predicted word
- ★ CONCAT – generates a phrase from verified word & its predicting phrase.

[Refer text if you need more details]

Consequences:

Blackboard approach has the following **Benefits:**

- ★ **Experimentation**
  - A blackboard pattern makes experimentation different algorithms possible.
- ★ **Support for changeability and maintainability**
  - Knowledge sources, control algorithm and central data structure are strictly separated.
- ★ **Reusable knowledge sources**
  - Knowledge source and underlying blackboard system understand the same protocol and data.
  - This means knowledge sources reusable.
- ★ **Support for fault tolerance and robustness**
  - Only those that are strongly supported by data and other hypotheses survive

Blackboard approach has the following **Liabilities:**

- ★ **Difficulty of testing**
  - Because it does not follow a deterministic algorithm
- ★ **No good solution is guaranteed**

- **Difficulty of establishing good control strategy**
  - We require an experimental approach.
- **Low efficiency**
  - Computational overheads in rejecting wrong hypothesis.
- **High development effort**
  - Most blackboard systems take years to evolve.
- **No support for parallelism.**

UNIT 4 – QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	With neat diagrams, depict the dynamic behaviour of pipes and filters pattern	Dec 09	10
2	What are the benefits of a layered pattern?	Dec 09	4
3	Give the structure of blackboard with CRC cards	Dec 09	6
4	What do you mean by architectural patterns? How is it categorized? Explain the structure part of the solution for ISO layered architecture	June 10	10
5	Explain with a neat diagram, the dynamic scenario of passive filters	June 10	10
6	List the components of a pipe and filters architectural pattern. With sketches, explain the CRC cards for the same	Dec 10	8
7	Explain the forces that influence the solutions to problems based on blackboard pattern	Dec 10	7
8	Write a note on the HEARSAY – II system	Dec 10	5
9	Discuss the 3-part schema which underlies the layers architectural patterns, with reference to networking protocols	June 11	14
10	Briefly explain the benefits offered by the pipes and filters pattern	June 11	6
11	Discuss the steps involved in the implementation of pipes and filters architecture	Dec 11	12
12	Write the context, problem and solution part of blackboard architectural pattern	Dec 11	8
13	List the components of a pipe and filters architectural pattern. With sketches, explain the CRC cards for the same	June 12	8
14	Define blackboard architectural pattern. Briefly explain the steps to implement the blackboard pattern	June 12	8
15	Write a note on the HEARSAY – II system	June 12	4