UNIT II

DESIGNING THE ARCHITECTURE WITH STYLES

DESIGNING THE ARCHITECTURE

ARCHITECTURE IN THE LIFE CYCLE

Any organization that embraces architecture as a foundation for its software development processes needs to understand its place in the life cycle. Several life-cycle models exist in the literature, but one that puts architecture squarely in the middle of things is the evolutionary delivery life cycle model shown in figure 2.1.

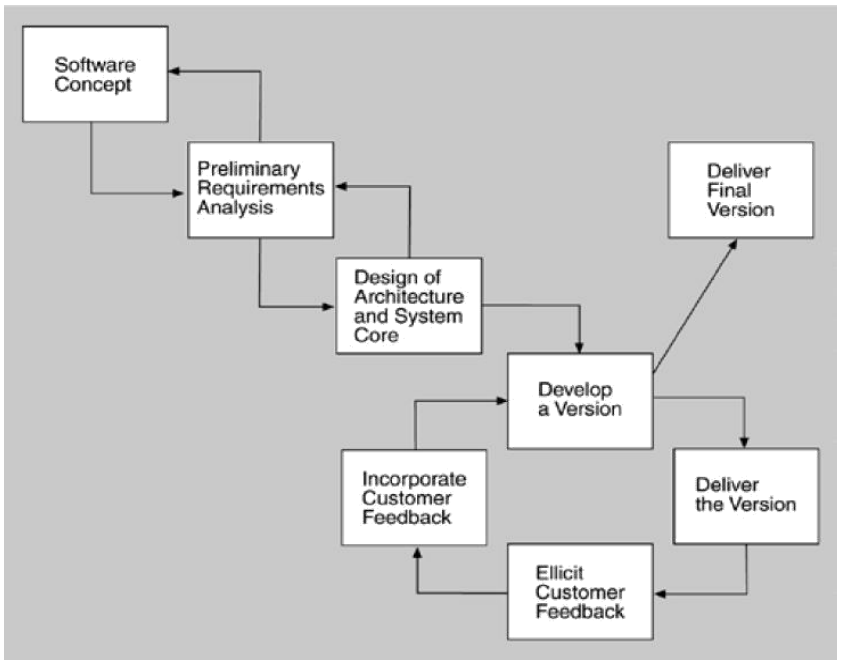
Software Concept
Preliminary Requirements Analysis
Design of Architecture and System Core
Develop a Version
Deliver Final Version
Deliver the Version
Ellicit Customer Feedback
Incorporate Customer Feedback

Figure 2.1. Evolutionary Delivery Life Cycle

The intent of this model is to get user and customer feedback and iterate through several releases before the final release. The model also allows the adding of functionality with each iteration and the delivery of a limited version once a sufficient set of features has been developed.

WHEN CAN I BEGIN DESIGNING?

The life-cycle model shows the design of the architecture as iterating with preliminary requirements analysis. Clearly, you cannot begin the design until you have some idea of the system requirements. On the other hand, it does not take many requirements in order for design to begin.

An architecture is “shaped” by some collection of functional, quality, and business requirements. We call these shaping requirements *architectural drivers* and we see examples of them in our case studies like modifiability, performance requirements availability requirements and so on.

To determine the architectural drivers, identify the highest priority business goals. There should be relatively few of these. Turn these business goals into quality scenarios or use cases.

DESIGNING THE ARCHITECTURE

A method for designing an architecture to satisfy both quality requirements and functional requirements is called attribute-driven design (ADD). ADD takes as input a set of quality attribute scenarios and employs knowledge about the relation between quality attribute achievement and architecture in order to design the architecture.

ATTRIBUTE DRIVEN DESIGN

ADD is an approach to defining a software architecture that bases the decomposition process on the quality attributes the software has to fulfill. It is a recursive decomposition process where, at each stage, tactics and architectural patterns are chosen to satisfy a set of quality scenarios and then functionality is allocated to instantiate the module types provided by the pattern.

The output of ADD is the first several levels of a module decomposition view of an architecture and other views as appropriate.

Garage door opener example

Design a product line architecture for a garage door opener with a larger home information system the opener is responsible for raising and lowering the door via a switch, remote control, or the home information system. It is also possible to diagnose problems with the opener from within the home information system.

Input to ADD: a set of requirements

Functional requirements as use
cases

- Constraints
- Quality requirements expressed as system specific quality scenarios

ADD Steps:

Steps involved in attribute driven design (ADD)

Choose the module to decompose

- Start with entire system
- Inputs for this module need to be available
- Constraints, functional and quality requirements

Refine the module

- Choose architectural drivers relevant to this decomposition
- Choose architectural pattern that satisfies these drivers
- Instantiate modules and allocate functionality from use cases representing using multiple views
- Define interfaces of child modules
- Verify and refine use cases and quality scenarios

Repeat for every module that needs further decomposition

Discussion of the above steps in more detail:

Choose The Module To Decompose

- the following are the modules: system->subsystem->submodule
- Decomposition typically starts with system, which then decompose into subsystem and then into sub-modules.
- In our Example, the garage door opener is a system
- Opener must interoperate with the home information system

Refine the module

Choose Architectural Drivers:

- choose the architectural drivers from the quality scenarios and functional requirements o The drivers will be among the top priority requirements for the module.
- In the garage system, the 4 scenarios were architectural drivers.

FORMING THE TEAM STRUCTURES

Once the architecture is accepted we assign teams to work on different portions of the design and development.

Once architecture for the system under construction has been agreed on, teams are allocated to work on the major modules and a work breakdown structure is created that reflects those teams.

Each team then creates its own internal work practices.

For large systems, the teams may belong to different subcontractors.

Teams adopt “work practices’ including

- Team communication via website/bulletin boards
- Naming conventions for files
- Configuration/revision control system.
- Quality assurance and testing procedure.

The teams within an organization work on modules, and thus within team high level of communication is necessary

CREATING A SKELETAL SYSTEM

- Develop a skeletal system for the incremental cycle.
- Classical software engineering practice recommends -> “stubbing out”
- Use the architecture as a guide for the implementation sequence
- First implement the software that deals with execution and interaction of architectural components
- Communication between components. Sometimes this is just install third-party middleware
- Then add functionality
  By risk-lowering Or by availability of staff

ARCHITECTURAL STYLES

List of common architectural styles:

Dataflow systems:
- ✓ Batch sequential
- ✓ Pipes and filters

Call-and-return systems:
- ✓ Main program and subroutine
- ✓ OO systems
- ✓ Hierarchical layers.

Independent components:
- ✓ Communicating processes
- ✓ Event systems

Virtual machines:
- ✓ Interpreters
- ✓ Rule-based systems

Data-centered systems:
- ✓ Databases
- ✓ Hypertext systems
- ✓ Blackboards.

PIPES AND FILTERS

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

Conditions (invariants) of this style are:

Filters must be independent entities. They should not share state with other filter . Filters do not know the identity of their upstream and downstream filters. Specification might restrict what appears on input pipes and the result that appears on the output pipes.

Correctness of the output of a pipe-and-filter network should not depend on the order in which filter perform their processing.

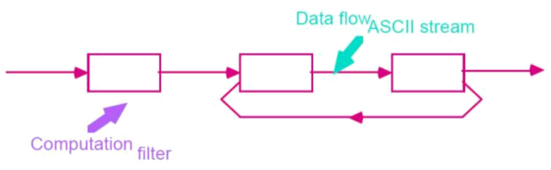

Figure 1: Pipes and Filters

Common specialization of this style includes :

Pipelines:

Restrict the topologies to linear sequences of filters.

Bounded pipes:

Restrict the amount of data that can reside on pipe.

Typed pipes:

Requires that the data passed between two filters have a well-defined type.

Batch sequential system:

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

Example 1:

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

Example 2:

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

Signal processing domains

Parallel processing

Functional processing

Distributed systems.

Advantages:

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

Disadvantages:

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

OBJECT-ORIENTED AND DATA ABSTRACTION

In this approach, data representation and their associated primitive operations are encapsulated in the abstract data type (ADT) or object. The components of this style are- objects/ADT's objects interact through function and procedure invocations.

Two important aspects of this style are:

Object is responsible for preserving the integrity of its representation.

Representation is hidden from other objects.

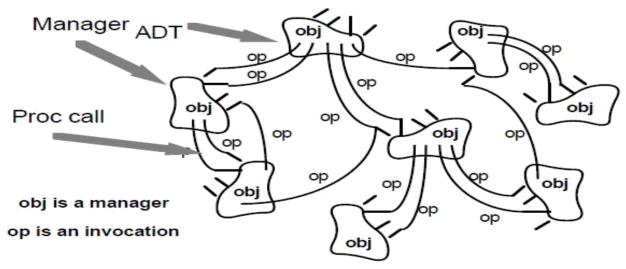

Figure 2: Abstract Data Types and Objects

Advantages

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

Disadvantages

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

EVENT-BASED, IMPLICIT INVOCATION

Instead of invoking the procedure directly a component can announce one or more events.

Other components in the system can register an interest in an event by associating a procedure to it.

When the event is announced, the system itself invokes all of the procedure that have been registered for the event. Thus an event announcement "implicitly" causes the invocation of procedures in other modules.

Architecturally speaking, the components in an implicit invocation style are modules whose interface provides both a collection of procedures and a set of events.

Advantages:

It provides strong support for reuse

Any component can be introduced into the system simply by registering it for the events of that system.

Implicit invocation eases system evolution.

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

Disadvantages:

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

LAYERED SYSTEMS:

A layered system is organized hierarchically.
Each layer provides service to the layer above it.
Inner layers are hidden from all except the adjacent layers. Connectors are defined by the protocols that determine how layers interact each other.
Goal is to achieve qualities of modifiability portability.

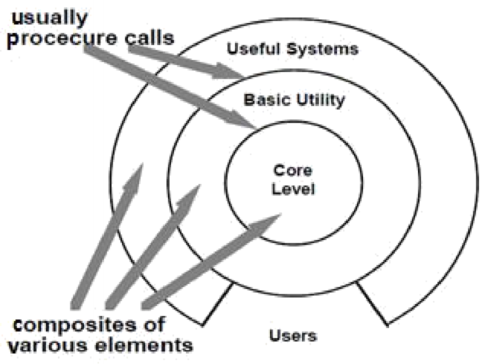

Figure 3: Layered Systems

Examples:

Layered communication protocol
Operating systems
Database systems

Advantages:

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

Disadvantages:

- Not easily all systems can be structures in a layered fashion.
- Performance may require closer coupling between logically high-level functions and their lower-level implementations.
- Difficulty to mapping existing protocols into the ISO framework as many of those protocols bridge several layers.
- Layer bridging: functions is one layer may talk to other than its immediate neighbor.

REPOSITORIES:[data cantered architecture]

Goal of achieving the quality of integrability of data. In this style, there are two kinds of components. Central data structure- represents current state. Collection of independent components which operate on central data store. The choice of a control discipline leads to two major sub categories.

- Type of transactions is an input stream trigger selection of process to execute
- Current state of the central data structure is the main trigger for selecting processes to execute.

Active repository such as blackboard.

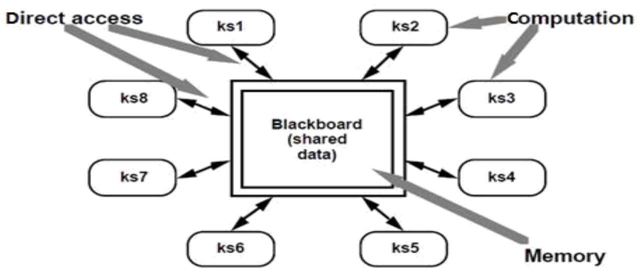

Figure 4: The Blackboard

Blackboard:

Three major parts:

- **Knowledge sources:**

  Separate, independent parcels of application – dependents knowledge.

- **Blackboard data structure:**

  Problem solving state data, organized into an application-dependent hierarchy

- **Control:**

  Driven entirely by the state of blackboard

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

The actual focus of control can be in

- knowledge source
- blackboard
- Separate module or
- combination of these

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

INTERPRETERS

An interpreter includes pseudo program being interpreted and interpretation engine.

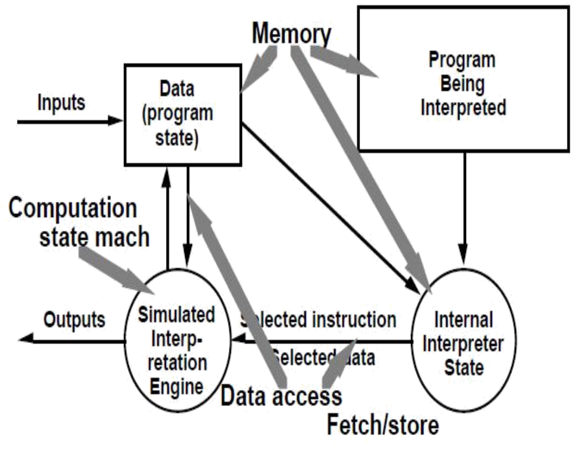

Figure 5: Interpreter

- Pseudo program includes the program and activation record.
- Interpretation engine includes both definition of interpreter and current state of its execution.
- Interpretation engine: to do the work
- Memory: that contains pseudo code to be interpreted.
- Representation of control state of interpretation engine
- Representation of control state of the program being simulated.

Ex: JVM or "virtual Pascal machine"

Advantages:

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

Disadvantages:

Performance cost because of additional computational involved