UNIT 2

Architectural Style

An architectural style consists of a set of component types (e.g., process, procedure, objects, and servers), a set of connectors (e.g., data streams, sockets) that mediate communication among components. It also includes a set of configuration rules. This could be in the form of some topological constraints that determine allowed compositions of elements. For example, a component may be connected to at most two other components. Overall, they provide a framework on which to base design.

LIST OF ARCHITECTURAL STYLES

1.PIPE AND FILTER
2. DATA ABSTRACTION AND OBJECT ORIENTATION
3. EVENT BASED IMPLICIT INVOCATION
4. LAYERED SYSTEMS5.INTERPRETERS
6.BLACK BOARD AND REPOSITORY
7.PROCESS CONTROL
8.HETROGENEOUS ARCHITECTURES

1. PIPE AND FILTER

NODES REPRESENT COMPONENTS
ARCS CONNECTORS
VOCABULARY OF COMPONENTS AND CONNECTORS WITH CONSTRAINTS
----------- MARY SHAW GARLAN

CONSTRAINTS ARE MAINLY WITH LOGIC ON WHICH APPLICATION WORKS.
BOUNDARIES OF STYLES OVERLAP

COMPONENTS ARE COMPUTING ELEMENTS --------- FILTERS
CONDUITS FOR STREAMS
--------- CONNECTORS

INVARIENTS

1.FILTERS ARE INDEPENDENT ENTITIES

2.FILTERS DO NOT KNOW THE IDENTITIES OF UPSTERAM AND DOWN STREAM FILTERS.

INVARIENTS OF PIPES

1. DEGENERATE : PROCESSES IN ONE SHOT AND DATA IS HANDLED AS A SINGLE ENTITY (BATCH SEQUENTIAL) AND PROGRAMS ARE WRITTEN IN UNIX.
2. UNIX PROVIDES RUN TIME MECHANISMS FOR IMPLEMENTING PIPES.

TRADITIONAL COMPILERS ARE PIPES AND FILTERS

1. LEXICAL ANALYSIS
2.PARSING
3.SEMANTIC ANALYSIS
4.CODE GENERATION

ADVANTAGES:

1. DESIGNER UNDERSTANDS OVERALL INPUT AND OUTPUT PROPERTIES.
2.SUPPORTS REUSE.
3.EASY ENHANCEMENT OF SYSTEM.
4.SUPPORTS THROUGHPUT AND DEADLOCK ANALYSIS.
5.SUPPORTS CONCURRENT EXECUTION.

DISADVANTAGES:

1. OFTEN LEADS TO BATCH SEQUENTIAL PROCESS(OPPOSITE OF INCREMENTAL PROCESS).
2.NOT GOOD AT INTERACTIVE APPLICATIONS.
3.PROBLEM OF HANDLING INCREMENTAL UPDATES.

4.DIFFICULTY IN MAINTAINING CORRESPONDANCE BETWEEN TWO SEPARATE BUT RELATED STREAMS.
5.PARSING AND UNPARSING IS REQUIRED FOR FILTERS SO THAT WRITING CODE FOR FILTERS IS DIFFICULT.

DATA ABSTRACTION AND OBJECT ORIENTATION

1.DATA REPRESENTATIONS AND OPERATIONS ARE EMBEDDED IN OBJECTS.
2.OBJECTS ARE COMPONENTS WHICH PRESERVES INTEGRITY OF REPRESENTATION.
3.OBJECTS INTERACT THROUGH FUNCTION INVOCATIONS.
4. REPRESENTATIONS ARE HIDDEN FROM OTHER OBJECTS.

INVARIANTS :
1. OBJECTS TO BE CONCURRENT TASKS
2. OBJECTS HAVE MULTIPLE INHERITANCES. ADVANTAGES:
1.IMPLEMENTATION CAN BE CHANGED WITHOUT AFFECTING OTHER OBJECTS.
2. DECOMPOSITION OF PROBLEM IS EASY.

DISADVANTAGES:
1.FOR ONE OBJECT TO INTERACT WITH ANOTHER OBJECT THEY SHOULD KNOW THE OBJECT IDENTITIES.
2. IF THE OBJECT IDENTITY CHANGES?

Event-based, Implicit Invocation Architectural Style

This is one of the more broadly accepted architectural styles. This architectural style is characterized by the style of communication between components. Rather than invoking a procedure directly or sending a message a component announces, or broadcasts, one or more events. The components in this style are independent reactive objects (or processes). These components have interfaces that provide a collection of procedures/methods and a set of events that it may announce. Some objects register interest in certain events and other objects that would signal those events. The automatic method invocation acts as connectors, by providing the binding between event announcements and procedure/method calls. The topologies are arbitrary in this style, with implicit dependencies arising from event announcements and registrations.

An example of this style can be seen in some Integrated Development Environment (IDE) where the Editor announces it has finished editing a module, and the compiler registers for such announcements and automatically re-compiles module. Another example in the same context is where the Debugger announces it has reached a breakpoint, editor registers interest in such announcements and automatically scrolls to relevant source line.

There are several advantages associated with this style. It provides strong support for reuse, because new components can be plugged in to the system by registering it for events. It also affords easy maintenance, because we can add and replace components with minimum affect on other components in the system.

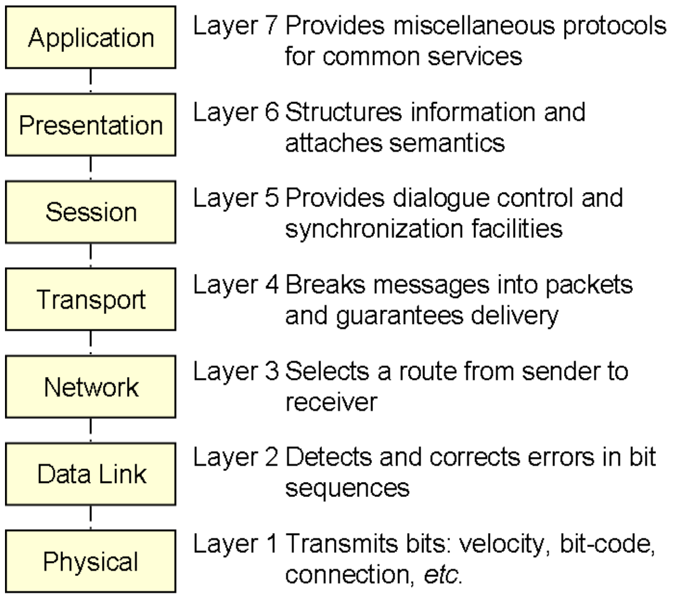
There are a few disadvantages with this system. First among them is the loss of control. For example, when a component announces an event, it has no idea what components will respond to it. It cannot rely on the order that these components will be invoked. Nor can it tell when they are finished. The non-deterministic nature of the execution sequence makes ensuring correctness difficult, because it depends on context in which invoked. Also you can have unpredictable interactions. Another issue is that exchange of data can require use of global variables or shared repository. This means that the task of resource management can become a challenge.

Layered Systems

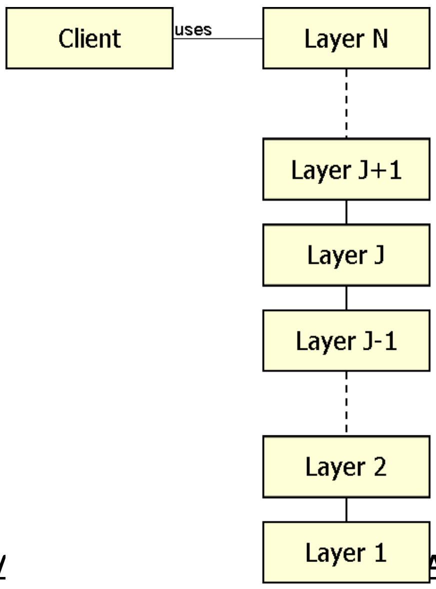
Layered systems have a hierarchical system organization. We can think of a layered system as a multi-level client-server. Each layer acts as a server, i.e., service provider to layers "above". It also acts as a client, .i.e., it is a service consumer of layer(s) "below".

Each layer exposes an interface (e.g., API or protocol) to be used by layer above it.

A popular example of a layered system is the OSI Seven-Layer model shown below


Application
Layer 7 Provides miscellaneous protocols for common services
Presentation
Layer 6 Structures information and attaches semantics
Session
Layer 5 Provides dialogue control and synchronization facilities
Transport
Layer 4 Breaks messages into packets and guarantees delivery
Network
Layer 3 Selects a route from sender to receiver
Data Link
Layer 2 Detects and corrects errors in bit sequences
Physical
Layer 1 Transmits bits: velocity, bit-code, connection, etc.

The generic structure of a layered system looks like this:



Here the client to the system or the user can interact only with the top-most layer. Request from the client/user travels down each layer, till reaches the bottom-most layer. where some function is performed, and the results of the service or function bubbles up in reverse order.

The advantages of the layered system are that it breaks down complex problem into incremental steps. Since interaction between layers is restricted to its adjacent layer, systems that use the layered style are usually easy to Modify. Also, because the style enforces standard layer interfaces, reusability of layers is enhanced.

The disadvantages of the layered system are that it is not suitable for all systems. It is not always easy to decompose a system into clean layers. Another issue is because communication can take place only between adjacent layers, data has to flow through several layers causing performance issues.

Data-centered Systems

A data-centered system is one in which all data is maintained in a central store, sometimes called a repository, and there exists a collection of external, independent components that operate on the data in the store.

There are two subcategories under this style.

First is the Traditional database, sometimes called repositories. Here the data is passive and independent components control the manipulations of the data store. Here, types of transactions trigger selection of processes to execute

Another subcategory is called Blackboard. Here the data is quasi-active, i.e., the data store informs the clients that interested in changes. The current state of the central data structure triggers computations in the independent components.

Blackboard

The blackboard architectural style originates in work in the Artificial Intelligence community. Hearsay II, the first continuous speech understanding systems developed in the 1970's by Erman, Hayes-Roth, Lesser, and Raj Reddy at Carnegie Mellon University used this style first.

We can understand the working of this style by imagining a group of specialists work cooperatively to solve a problem, using a blackboard as the workplace for developing the solution. The problem and initial data are written on the blackboard. The specialists watch the blackboard, and when a specialist finds sufficient information on the board to make a contribution, he records his contribution on the blackboard.

This style is useful in an immature domain in which no closed approach to a solution is known or feasible, and to solve it requires multiple distinct kinds of expertise. Since there is no deterministic algorithm, there are always many options for execution sequence. Also there is a great deal of uncertainty. i.e., the domain suffers from error and variability in both input data and knowledge. Blackboard style can be used where "Best effort" or approximate solution is good enough, because this is better than no solution.

Since this style will likely be used in an immature domain, experimentation in algorithm necessary. That means that all the modules within the system should be exchangeable.

The solution this style proposes is to have a collection of independent program working cooperatively on a common data structure. Each program is a specialist that solves a part of the problem. They do not interact with each other. Another characteristic is that there is no predetermined sequence of operation. At any point of time, the current state of data determines what happens. There will be a central control component that evaluates state and coordinates the specialist modules.

We have to view each step as a problem-solving process, where partial solutions are constantly combined, changed or rejected. The solution space is organized into levels of abstraction.

Structure in Blackboard Style

There are three components: Blackboard data structure, Knowledge sources, and Central Control.

The Blackboard data structure stores the entire state of problem solution. The data structure is hierarchical and non-homogeneous. The knowledge sources are separate, independent subsystems that solve specific aspects of the overall problem. The knowledge sources do not communicate directly. They can only read and write from the blackboard. The Control component runs a loop that monitors change on the blackboard data structure and decides what to do using some strategy to make knowledge sources respond opportunistically.

Benefits

This styles major benefit is that it allows for experimentation. It allows different algorithms to be tried, and, different control heuristics to be used. The positive side effect of the knowledge sources not communicating directly is that this style supports changeability and maintainability very well.

Disadvantages

Since the style is non-deterministic in nature, the system becomes difficult to test. Also, the style does not guarantee a good solution. Often in implementations there is difficulty of establishing a good control strategy. Also, the style is basically is not very efficient because there are a lot of computational overheads. The development effort needed is also very high, which is mainly due to the ill-structured problem domains.

CASE STUDY

1. KWIC :

Reasons for using kwic are

MOTIVATION FOR CHANGE
1. CHANGE IN ALGORITHM
2. CHANGE IN DATA REPRESENTATION
3. ENHANCEMENT TO FUNCTION
4. PERFORMANCE – SPACE AND TIME
5. REUSE

KEY WORD IN CONTEXT (KWIC)
example components chosen are

INPUT
CIRCULAR SHIFT
ALPHABETIZER
OUTPUT

example connectors chosen are

DMA
SUBPROGRAM CALL
SYSTEM I/O

The architectural styles used are

A. MAIN PGM/SUBROUTINE WITH SHARED DATA

1. FUNCTIONAL DECOMPOSITION WITH SHARED ACCESS TO DATA
2. DECOMPOSITION THAT HIDES DESIGN DECISIONS
REUSE DIFFICULT

B. ABSTRACT DATA TYPE

DECOMPOSITION --- EACH MODULE PROVIDES AN INTERFACE WHICH PERMITS OTHER COMPONENTS TO ACCESS DATA BY INVOKING PROCEDURES IN THAT INTERFACE
NOT SUITED FOR FUNCTIONAL ENHANEMENTSCHARACTER
CIRCULAR SHIFT
ALPHABETIC SHIFTIMPLICIT INVOCATION
ACCESS DATA AS A LIST OR SET

COMPUTATIONS ARE INVOKED IMPLICITLY(SELECTIVE BROADCAST)
DIFFICULTY IN CONTROL

C. IMPLICIT INVOCATION

COMPONENT ANNOUNCES ONE OR MORE EVENTS ,
OTHER COMPONENTS REGISTERS BY ASSOCIATING A PROCEDURE WITH IT.
WHEN EVENT IS ANNOUNCED ALL THE PROCEDURES THAT WERE REGISTERED ARE INVOKED

D. PIPE AND FILTER

MODIFICATION DIFFICULT

2.INSTRUMENTATION SOFTWARE

A. OBJECT ORIENTED MODEL
OBJECTS WHICH ARE CHOSEN ARE AS FOLLOWS

OSCILLOSCOPE OBJECT
WAVEFORM
MAXMIN WAVEFORM
X-Y WAVEFORM
DISADVANTAGE IS DIFFICULTY IN FUNCTIONALITY PARTITIONING

LAYERED

Layers which are chosen are as follows

HARDWARE
DIGITIZATION
VISIUALISATION
UI
MANIPULATION

DISADVANTAGE IS BOUNDARIES OF ABSTRACTION DIFFICULT

PIPE AND FILTER

INCREMENTAL TRANSFORMATION OF DATA CONTROL BY EACH FILTER
DISADVANTAGE IS POOR PERFORMANCE

MODIFIED PIPE AND CONTROL

INDIVIDUAL FILTER INPUT CONTROL

component may be connected to at most two other components. Overall, they provide a framework on which to base design.

Interpreters:

In interpreter organization a virtual machine is produced in software. It includes pseudoprogram and interpreter engine.

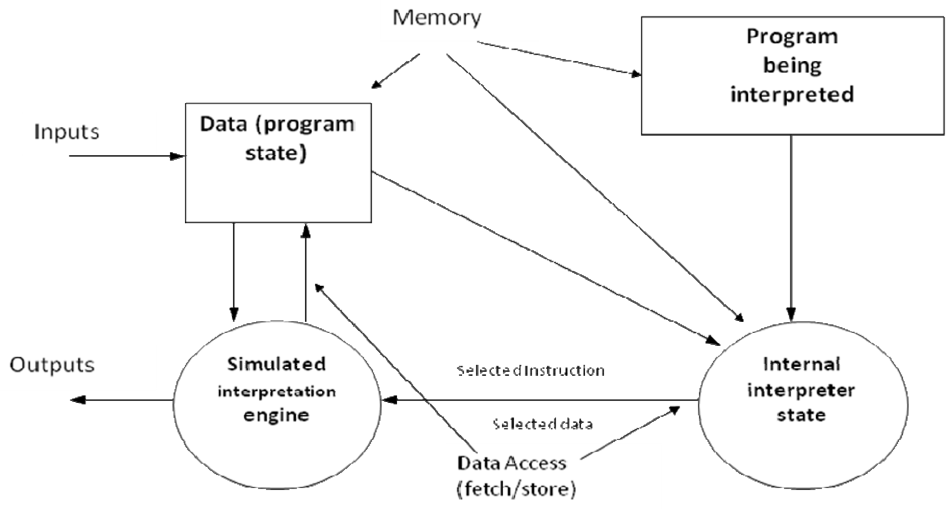
Pseudoprogram has:

- Program itself
- Interpreter's analog of its execution state

Interpretation engine is characterized by

- Definition of the interpreter
- Current state of its execution

The following diagram shows the different components and connectors of an Interpreter Architectural Style:


Memory
Program being interpreted
Inputs
Data (program state)
Outputs
Simulated interpretation engine
Selected instruction
Selected data
Internal interpreter state
Data Access (fetch/store)

It is commonly used to build virtual machines that close the gap between the computing engine expected by the semantics of program and the computing engine available in hardware.

Process Control:

Unlike other models, Process control is characterized by both kinds of components involved and relations that hold among them.

Continuous processes convert input materials to products with specific requirements by performing operations on the input and intermediate products.

It has related terminologies which we call Process Control Paradigms:

Process variables: The values of the measurable properties of the system state

Controlled variables: the process variables that measure the output materials

Input variable: Process variable that measures the input

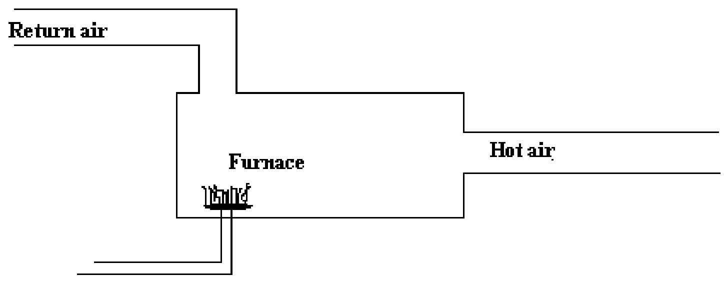
Manipulated variable: process variable whose value can be changed by the controller

Set point: the desired value for a controlled variable

Open-loop system:

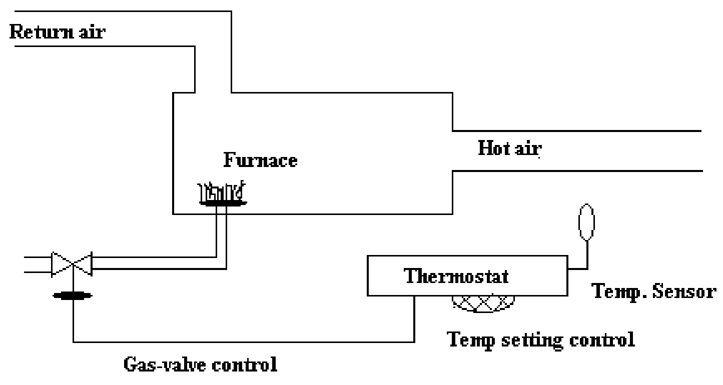
If the input materials are pure, process is fully defined and the operations are completely repeatable, then the process can run without surveillance.

Consider the following example where there is a hot-air furnace that uses a constant burner setting to raise the temperature that passes through the furnace.



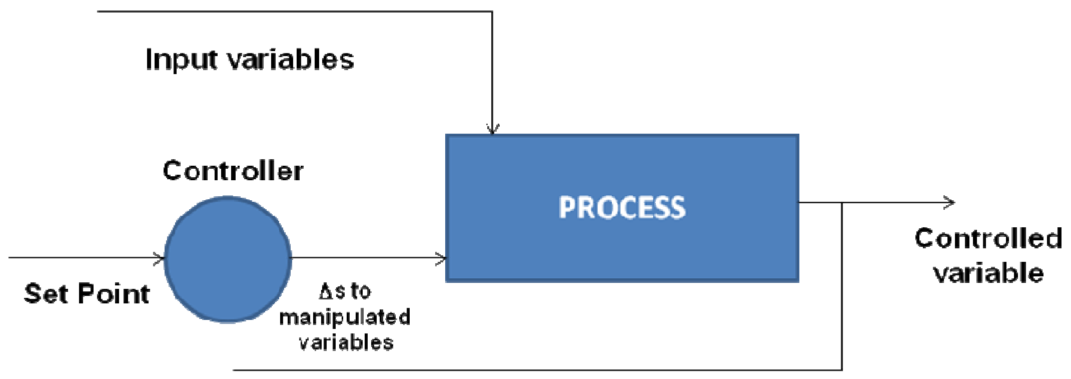
Closed Loop system:

Open loop assumptions are rarely valid in real world. More often the properties are monitored and their values are used to control the process. Such systems are called closed-loop systems. A home thermostat is a common example. The air temperature at the thermostat is measured and the furnace is turned on or off as necessary to maintain the desired temperature.



There are two categories of closed-loop control:

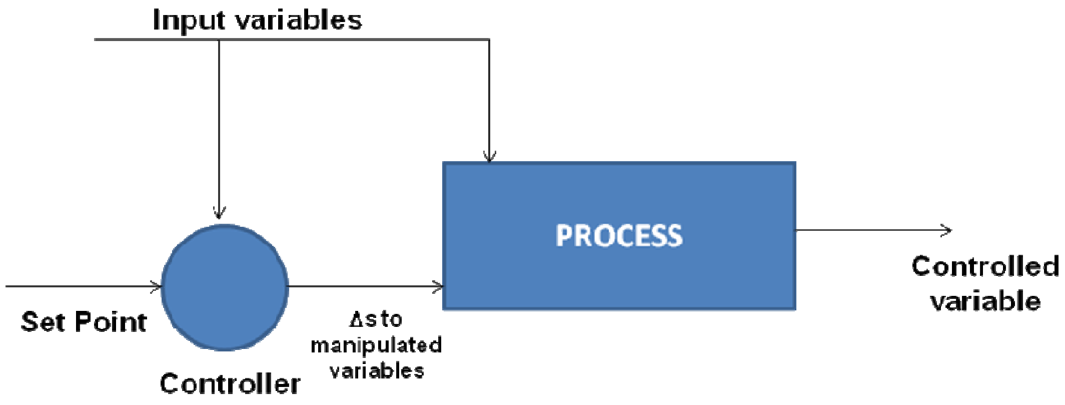
Feedback Control - Adjusts the process according to the measurement of controlled variable



The important components are process variables, a sensor to obtain the controlled variable from the physical output, the set point and a control algorithm.

Feed forward Control

Anticipates future effects on the controlled variable by measuring other process variables whose values may be more timely



The components are essentially the same as in feedback controller except that the sensor obtains the values of input or intermediate variables.

A software paradigm for Process Control

- We think of software as algorithmic
- Outputs are solely based on inputs
- This model does not allow for external perturbations
- This normal software model corresponds to open-loop systems
- When operating condition of software systems are non predictable, pure algorithmic model breaks down
- When the execution of a software is affected by external disturbances, a control paradigm should be considered

Process-control model can be used as architectural style for the software that controls the continuous process.

Essential parts of that loop would be

- Computational Elements: separate the process of interest from the control policy
  - Process definition: mechanism for manipulating variables
  - Control algorithm: to decide how to manipulate variables and models how process variables reflect true state
- Data Elements: updated process variables and sensors
  - Process variables: inputs, controlled and manipulated variables
  - Set point: reference value of controlled variable

Sensors: to obtain the value of process variables

- The control loop paradigm
  - collects the information about actual and intended states of the process
  - Tunes process variables to drive the actual state towards intended state
- The two computational elements separate issues about functionality from issues about responses to external disturbances

- We can bundle process and process variables ie. Process definition with process variables and sensor will form one subsystem
- Control algorithm and set point will form second subsystem

Other Architectures

- **Distributed processes**
  - DS have created a number of organizations for multiprocessor systems
  - Some are characterized by their topological features such as ring and star
  - Others are characterized in terms of inter-process protocols used for communication
  - Client server is the common distributed architecture
  - Sever provides services to other processes
  - Server does not know in advance the number or identities of the clients
  - But clients know the identity of server
- **Main program/subroutine organizations**
  - Primary organization of the system mirrors the programming language in which it is written
  - For languages without support for modularization, this often results in systems organized around main program and a set of subroutines
  - Main program acts as driver for subroutine providing control loop to sequence through in an order
- **Domain-specific architectures**
  - These are reference architectures for specific domains
  - They provide organizational structures tailored to a family of applications such as avionics, vehicle management systems
  - It is possible to increase the descriptive power of the system by these domain specific architectures
  - Sometimes an executable system can be automatically or semi automatically generated by these descriptions
- **State transition systems**
  - A common organization for many reactive systems
  - They are defined in terms of a set of states and transitions

Heterogeneous Architectures

While it is important to understand nature of individual styles, most systems use combination of architectural styles. There are three ways to combine styles:

One way is through hierarchy.

- A component organized in on architectural style may have an internal structure developed completely using other style – Unix pipes
- Even connectors too can be hierarchically decomposed

Second way is to allow single component to use mixture of connectors – repository through interpreter style, other components through pipes

Third way – elaborate one level in different architectural style.

Mobile Robotics

Mobile robotics refers to any system that controls a manned or partially manned vehicle, like, Car, submarine, space vehicle, etc. The software to control the robot will get input from external sensors and will control the actuators, usually in real-time, to control the motion of the robot or some tools that is part of the robot. An important higher-level function will be to plan the future path of the robot.

There are various complicating factors in the performance of the tasks, like there might be obstacles may block the path, or the sensor input might be imperfect. The robot might run out of power or fuel. It might be difficult to assure the accuracy in movement, probably due to some slippage of the wheels, etc. Some robots might to have to perform manipulation with hazardous material. Unpredictable events might lead to need for rapid response.

Before evaluating the solutions that use different architectural style, let us specify the criteria by which to evaluate them.

Requirement #1: The architecture must accommodate deliberative and reactive behavior. The robot has to coordinate the actions it deliberately undertakes to achieve its designated objective (e.g., collect a sample of rocks) with the reactions forced on it by the environment (e.g., avoid an obstacle).

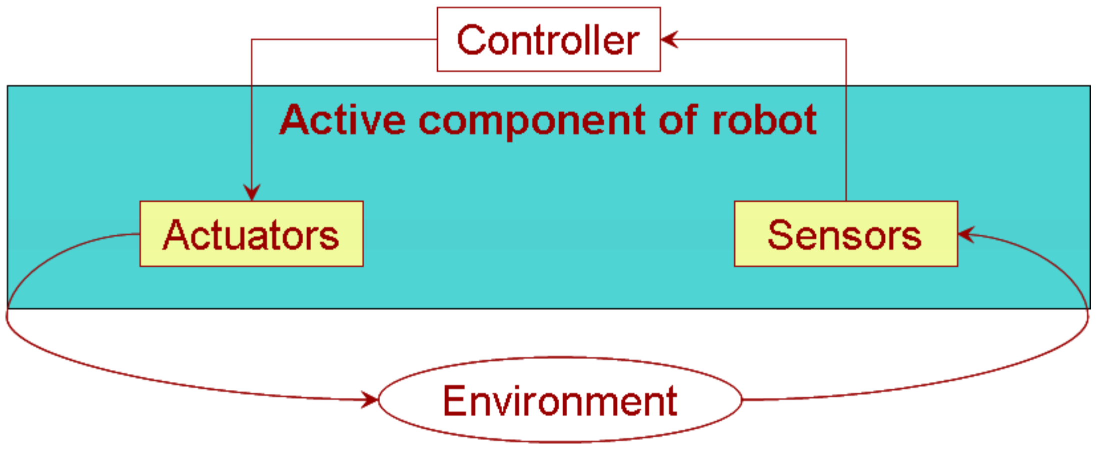
Requirement #2: The architecture must allow for uncertainty. Never will all the circumstances of the robot's operation be fully predictable. The architecture must provide the framework in which the robot can act even when faced with incomplete or unreliable information (e.g., contradictory sensor readings).

Requirement #3: The architecture must account for the dangers inherent in the robot's operation and its environment. By incorporating consideration of fault tolerance (R3a), safety (R3b), and performance (R3c) attributes, the architecture must help in maintaining the integrity of the robot, its operators, and its environment. Problems like reduced power supply, dangerous vapors, or unexpectedly opening doors should not spell disaster.

Requirement #4: The architecture must give the designer flexibility. Application development for mobile robots frequently requires experimentation and reconfiguration. Moreover, changes in tasks may require regular modification.

One thing to note of course is that the priority of these requirements will vary with the complexity of task and the unpredictability of its environment. For example, if the robot is to operate in Mars, fault tolerance becomes very important.

The first solution we will look at uses the **Control Loop paradigm**.



The first requirement we will use evaluate the solution is the need for deliberate and reactive behavior. The simplicity of the paradigm makes it a good solution for handling deliberate behavior. By the same token, the simplistic nature of the solution makes it very unsuitable in unpredictable environment. The control loop paradigm makes an implicit assumption that the continuous changes in environment require continuous reaction. But robots would face discrete events. So, it is very difficult with this paradigm to switch between behavior modes of handling deliberate and reactive behavior.

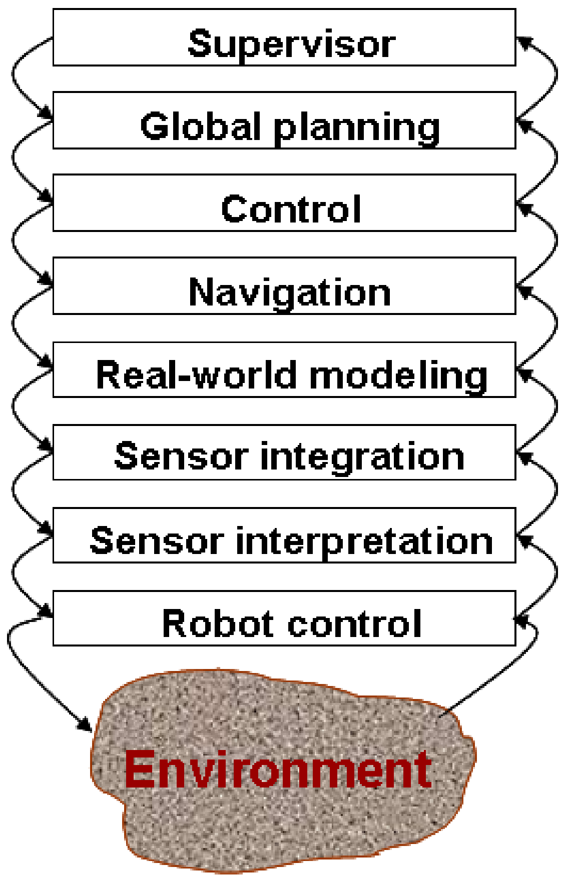
The next requirement is the allowance for uncertainty. In this paradigm uncertainty is resolved by reducing unknowns through iteration, which is basically a trial and error process. If more sophisticated steps are needed, this architecture offers no framework.

The third requirement has to do with handling dangers in the operations and in the environment. The control loop style fares well in this criterion. Fault tolerance and safety are enhanced by the simplicity of the architecture.

The last requirement of flexibility is also met by this style. Major components, like the supervisor, sensors, and actuators are separated and can be replaced independently. More refined tuning must take place inside the modules.

To summarize, this architectural style might be appropriate for simple robotics, as it can handle only a small number of external events. Clearly this is not really for complex decomposition of tasks.

The next style we will look at is the **Layered** architecture, which has been used as the basis for architecture of Terregator and Neptune mobile robots.



Let us look at the different layers here. The first layer performs the robot control routines, which includes moving the motors, and other tools. This layer will read the sensor inputs that will sense the relevant facets of the environment, like temperature, vision, etc. The second layer will perform the function of sensor interpretation, which is analysis of data from the various sensors. The third layer is the sensor integration layer, which performs combined analysis of different sensor inputs. The next layer above is concerned with maintaining the robot's model of the world. Layer five manages the navigation of the robot. Layer six will schedule the robot's actions. Layer seven will plan and deal with problems and replanning. The top level provides user interface and overall supervisory functions.

Evaluation of the architectural style:

The first criteria we will look at are the support for deliberate and non-deliberate or reactive behavior. This style sidesteps some of the problems encountered with the control loop, by defining more components to which the required tasks can be delegated. It also indicates the concerns that must be addressed (e.g., sensor integration). It also defines abstraction levels (e.g., robot control vs. navigation) to guide the design. But the problem is that layers suggest that services and requests are passed between adjacent components, which are not practical. Data requiring fast reaction may have to be sent directly from the sensors to the problem-handling agent at level 7, and the corresponding commands may have to skip levels to reach the motors in time. Another problem is that the layered architecture does not separate the two abstraction hierarchies that actually exist in the architecture. We have the data hierarchy is from layer 1 through 4, with raw sensor input, interpreted and integrated results, and finally the world model is built. The control hierarchy starts at the topmost layer and goes down through layers 7, 6, 5, and then directly to layer 1.

The second requirement of resolution of uncertainty is handled very well here. The abstraction layer addresses the need for managing uncertainty. What is uncertain at a lowest level may become clear in the higher layers. The conflicting sensor data can introduce ambiguity, but, the context embodied in the world model can resolve it.

The third requirement of Fault Tolerance is addressed partly by this architecture. The abstraction layer addresses the need for fault tolerance and passive safety, when you strive not to do something. Data and commands are analyzed from different perspectives. This style particularly allows us to incorporate many checks and balances, which is essential for satisfying this requirement. But, active safety poses a problem. When you have to do something rather than avoid doing something would require the communication pattern be short-circuited, which would violate the constraint in the layered architectural style of allowing only adjacent layers to communicate with each other.

The last requirement of flexibility is not met here, because of the tight coupling between layers. It is difficult to make replacement and addition of components due to that. Another issue is that the relationships between the layers are complex.

In sum, positive aspect is that the abstraction levels defined by the layered architecture provide a framework for organizing components. The framework succeeds because it is precise about the roles of the different layers. But, there are major drawbacks in this, too. The model breaks down when taken to the greater level of detail demanded by an actual implementation. Also, the communication patterns in a robot are not likely to follow the orderly scheme implied by the architecture.

The next architectural style we will evaluate is the **Task Control Architecture** (TCA) which uses **implicit invocation**. Task-specific modules communicate through API library that accesses a general-purpose reusable central control module. Control is centralized but data and problem solving is distributed.

TCA is based on a hierarchy of tasks called task trees, which are directed at one or more tasks registered to handle them. The parent tasks will initiate subtasks, which will be the child tasks. The task trees also allows for specifying temporal dependencies between tasks, allowing selective concurrency. If an exception occurs, then dynamic reconfiguration of task trees can be done at run time.

Three important features of this style make it ideal for this domain. Certain conditions cause the execution of an associated exception handler. Exceptions override the currently executing task in the subtler that causes the exception. They quickly change the processing mode of the robot and are thus better suited for managing spontaneous events (such as a dangerous change in terrain) than the feedback loop or the long communication paths of the pure layered architecture.

Another feature is called Wiretapping, which allows for interception messages. For example, a module wiretap certain messages to perform safety-checks of outgoing commands.

The third feature is called Monitors, which allows modules to read information and execute actions if the data fulfill a certain criterion. This feature allows for having specialist modules that will handle fault-tolerance in the system that will supervise the system. For example, one module can monitor the battery levels, and if it falls below a threshold, it can take corrective action, like issuing orders to go to base station to charge the batteries.

Let us now look at how this architectural style performs in the four criteria we had seen earlier.

The first requirement was deliberative and reactive behavior. The separation of action and reaction in the task trees by providing for explicit support for exceptions provides strong support for this requirement. TCA exception handling allows error recovery algorithms to be added without modifying/complicating the original algorithms (i.e., provides program extensibility through incremental, modular adjustments). The other two features of the TCA, wiretapping and monitors, helps in having dedicated modules that can handle emergencies. Another strong characteristic of this style is that the style provides explicit support for concurrency. But, in practice the parallelism might be limited by the capabilities of the central server. Another negative is the reliance on a central control point can be a weakness.

Although the second requirement of dealing with uncertainty is not supported explicitly in the model. But, it is possible to do this through task trees and exceptions.

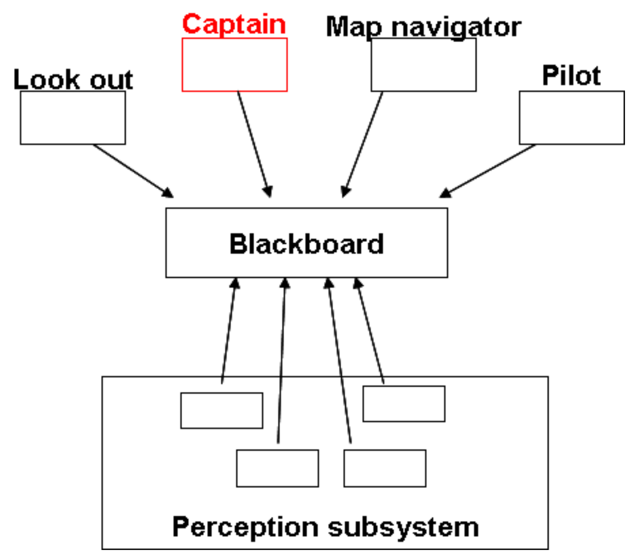
The third requirement has to do with handling dangers in the operations and in the environment. Exceptions, wiretapping and monitors works towards performance, safety and fault tolerance. Also, fault tolerance by redundancy is achieved by allowing multiple handlers for same signal concurrently.

The last requirement of flexibility is well supported by implicit invocation, which allows incremental development and replacement of components. It is usually sufficient to register new handlers, exceptions, wiretapping and monitors with central control, and the existing components are not affected.

So, what we find is that the TCA architecture, along with implicit invocation offers a comprehensive set of features for coordinating tasks of robot based on both expected and unexpected events. It allows modules to focus on independent separate tasks. TCA also provides better support for autonomous behavior of parts than layered architecture. Although the implementation of this style is not very straightforward, it is certainly appropriate for complex robotic projects

The last architectural style we will evaluate is the **Blackboard** architecture style, which was used in the Navlab project (computer-controlled vehicles for automated and assisted driving).

The various components in the system are the Caption, which acts as the overall supervisor. Then you have the map navigator, which performs high-level path planning. There is also the Lookout for monitoring the environment. The Pilot acts as the low-level path planner and motion controller. Lastly, there is the Perception subsystem which gets input from multiple sensors and integrates it into a coherent interpretation.



First requirement is the need to handle both deliberate and reactive behavior. Components interact via shared repository, so multiple agents can deal with same event. So, we can have one agent that deals with deliberate behavior. At the same time the Lookout can at the same time be monitoring the environment, and can initiate reactive behavior, when the state changes. One difficulty with this architecture is that all control flow has to be coerced to fit the database mechanism, even under circumstances where direct interaction between components would be more natural.

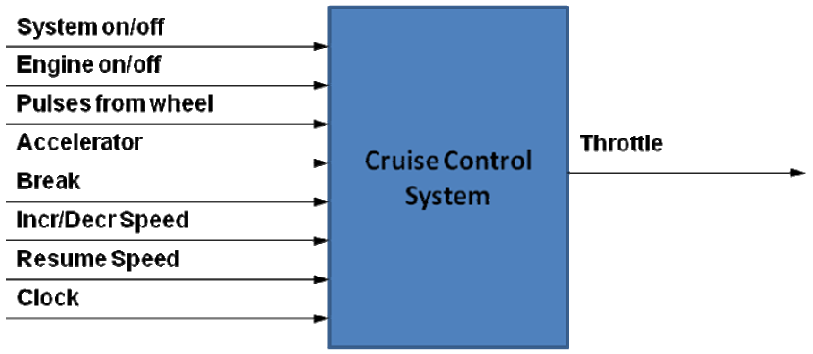
The allowance for uncertainty can be handled directly by the blackboard, which can resolve conflicts and uncertainty.

Case Study – Cruise Control

Booch and Others defined the problem as:

A Cruise-control System that exists to maintain the speed of the car, even over varying terrain

The hardware is as follows:



Problems with the original definition:

- Does not clearly state the rules for deriving outputs from inputs
- Booch's dataflow diagram fails to answer some questions
- Problem statement says o/p is a value for *throttle setting*
- Actually, It is *change in the throttle setting*
- *Current speed* is not explicit in the statement
- If it is addressed, then *throttle setting* is appropriate
- Also specifies, millisecond clock
- Used only to determine the current speed
- Number of clock pulses between wheel pulses are counted
- Problem is over specified in this respect
- A slower clock is sufficient and requires less computing
- Further, there is no need to use single clock for the entire system

Modified Definition is as follows:

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

Control Loop architecture is appropriate when the software is embedded in a physical system that involves continuing behavior especially when the system is subject to external perturbations.

These conditions hold in the case of cruise control – the system is supposed to maintain the constant speed in automobile despite variation s in terrain, vehicle load, air resistance etc.

Here, Computational Elements

- Process definition: The process receives the throttle setting and control the speed of the vehicle

- Control Algorithm: Models the current speed from wheel pulses, compares it with the desired speed and changes the throttle setting.
- Data Elements:
  - Controlled Variable – current speed
  - Manipulated Variable – throttle setting
  - Set point – desired speed
  - Sensor

The restated control task divides the problem into two sub problems:

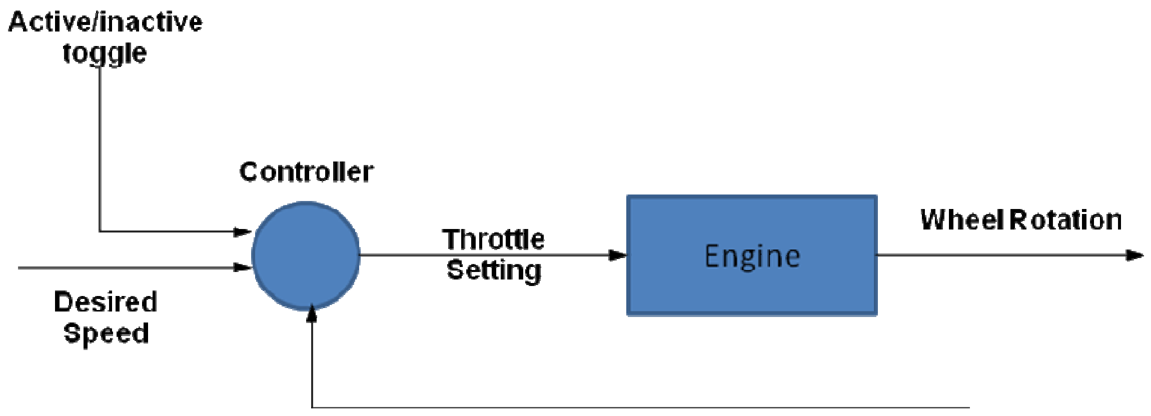
- Interface with the driver
- Control Loop

First task :

– model current speed from wheel pulses. The model could fail if the wheels spin

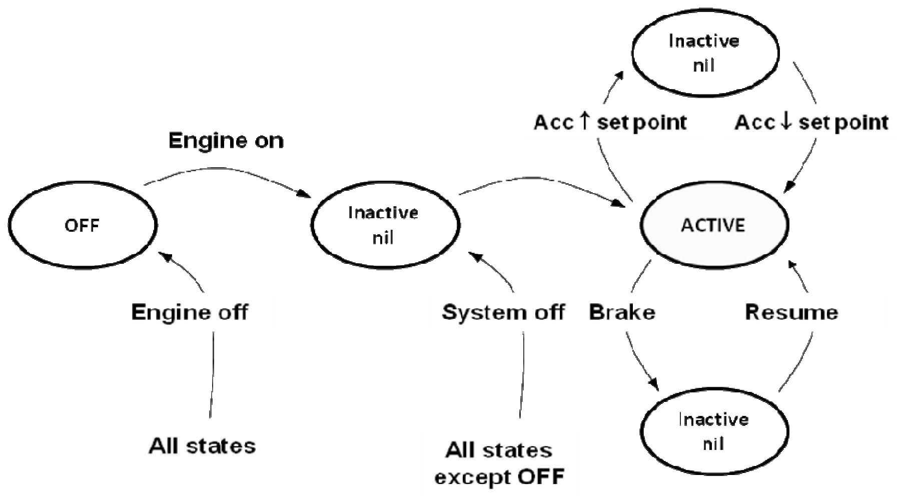
- If the pulses are taken from the drive wheel and is spinning, the cruise control would keep the wheel spinning, even if the vehicle stops moving.
- If the pulses are taken from nondrive wheel, and if drive wheel is spinning, the controller will act as if the current speed is too slow and continually increase the throttle setting.
- Further, there is no a full control authority over the process
  - In this case, the only manipulated variable is throttle. If the automobile is coasting faster than the desired speed, the controller is powerless.

The control Architecture:



- The controller receives two inputs

  - Active/inactive toggle: indicates whether the controller is in charge of throttle
  - Desired speed: needs to be valid only when the vehicle is under automatic control
- All these should be either state or continuously updated data, so that all lines represent data flow
- Controller is implemented as a continuously evaluating function
- Several implementations are possible – simple on/off control, proportional control etc
- The set-point calculation has two parts:
  - Determining whether or not the automatic system is active
  - Determining the desires speed for use by controller
- Some inputs in the defn capture state and others capture events.
- However, to work with the automatic controller, everything it depends on should be same
- Therefore, transitions are used between states.



The above figure shows the state machine for activation.

- System is completely off when engine is off
- There are 3 inactive states and 1 active state
- No set-point is established in the first inactive state
- Set-point is to be remembered in the remaining two
- The active/inactive toggle input of the control system is set to active when this state machine is in active state

Steps to determine the desired speed

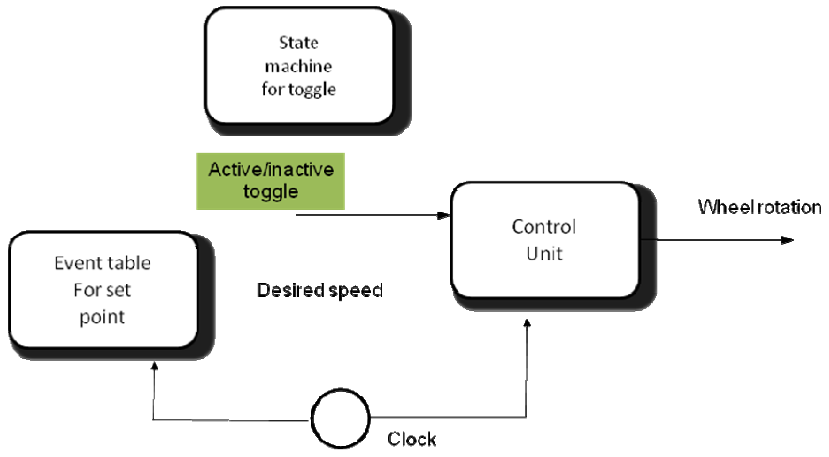
- Only current value of the desired speed is needed
- Whenever the system of off, set-point is undefined
- Whenever the system is switched on
  - Set-point will take the value from current speed modeled by wheel pulses
  - Driver also has a control that inc/decr set-point value by a set amount

Event	Effect on desired speed

Engine/system off	Set to “undefined”
System on	Set current speed using esti. from wheel pulses
Increase speed	Incr. desired speed by constant
Decrease speed	Decr. Desired speed by a constant

This is the event table for determining the set point.

Complete Cruise Control System



- All the objects of Booch’s design have clear roles
- The shift raises a number of questions which were slighted in the early design
- The separation of process from control concerns led to explicit choice of control discipline
- The limitations also became clear – possible inaccuracies in the current speed model and incomplete control at higher speed.