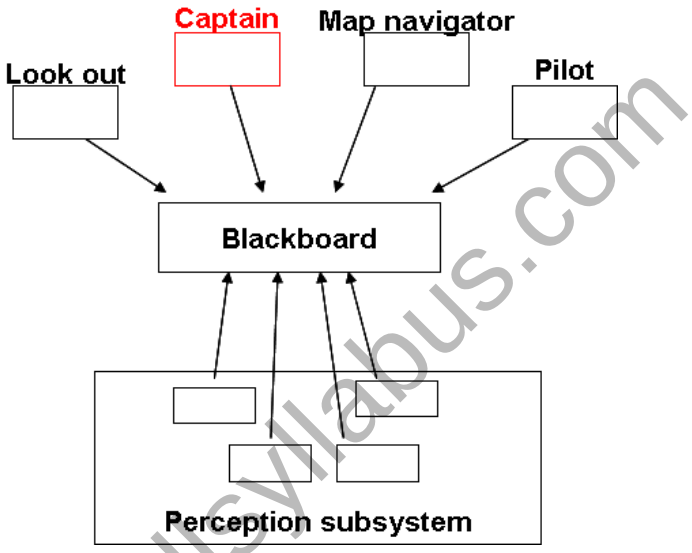
The last architectural style we will evaluate is the **Blackboard** architecture style, which was used in the Navlab project (computer-controlled vehicles for automated and assisted driving).

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.

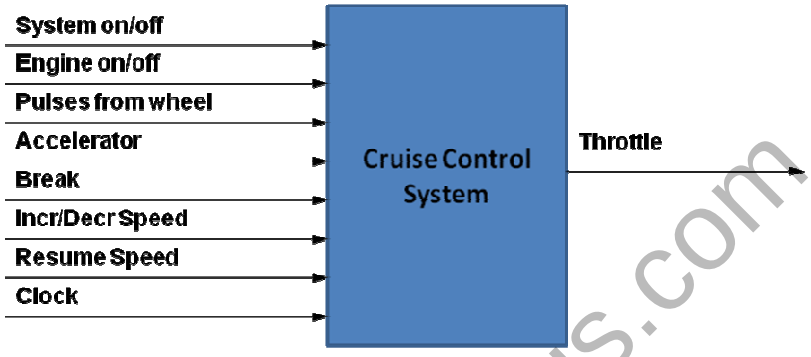
The allowance for uncertainty can be handled directly by the blackboard, which can resolve conflicts and uncertainty.

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:

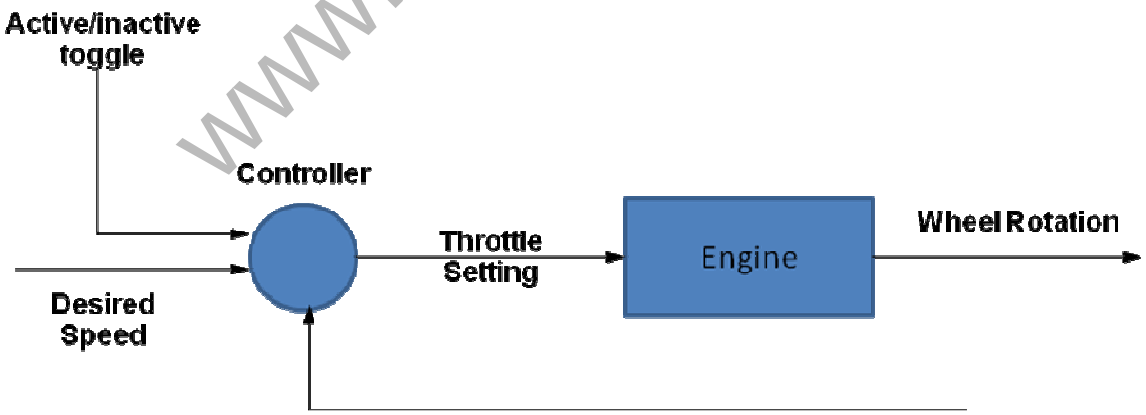
- Interface with the driver
- Control Loop

First task :

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

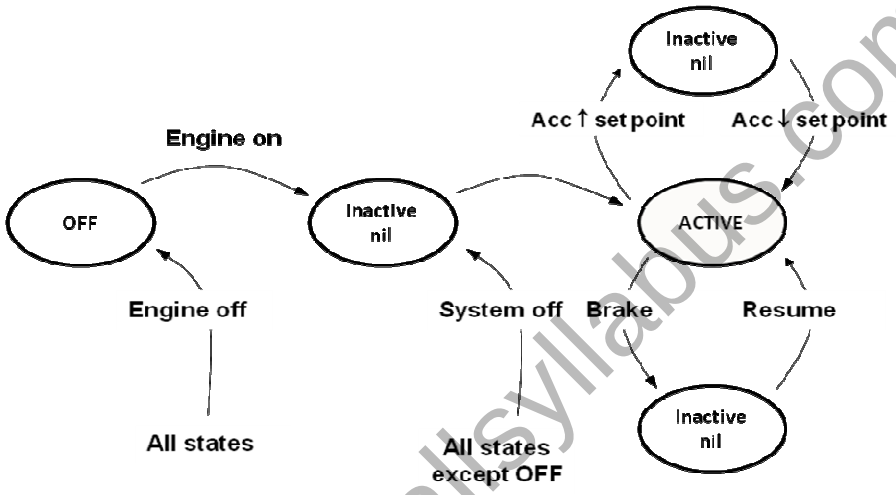
- If the pulses are taken from the drive wheel and is spinning, the cruise control would keep the wheel spinning, even if the vehicle stops moving.
- If the pulses are taken from nondrive wheel, and if drive wheel is spinning, the controller will act as if the current speed is too slow and continually 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

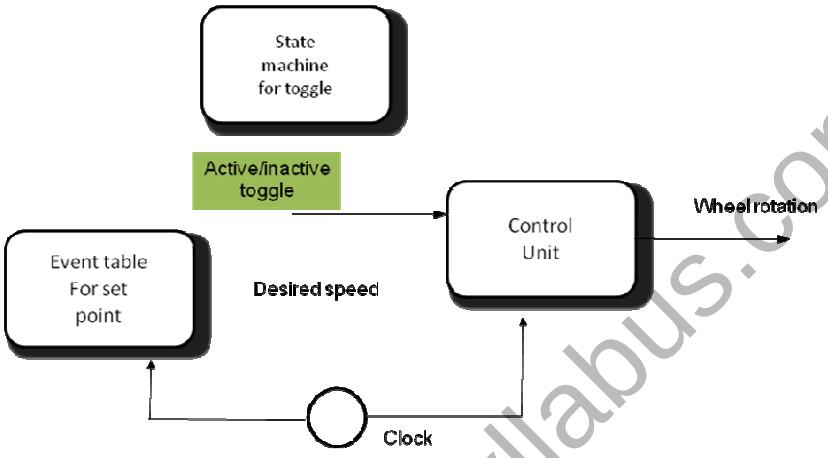
- Only current value of the desired speed is needed
- Whenever the system of off, set-point is undefined
- Whenever the system is switched on
  - Set-point will take the value from current speed modeled by wheel pulses
  - Driver also has a control that inc/decr set-point value by a set amount

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.

UNIT 3

Understanding Quality Attributes

Quality must be considered at ***all*** phases of design, implementation, and deployment
Architecture is critical to the realization of many of the qualities of interest in a system, and these qualities should be designed in and evaluated at the architectural level
Some qualities are not architecturally sensitive and attempting to achieve these qualities or analyze for them through architectural means is not fruitful
Quality attributes exist within complex systems and their satisfaction can never be achieved in isolation .

Quality attributes

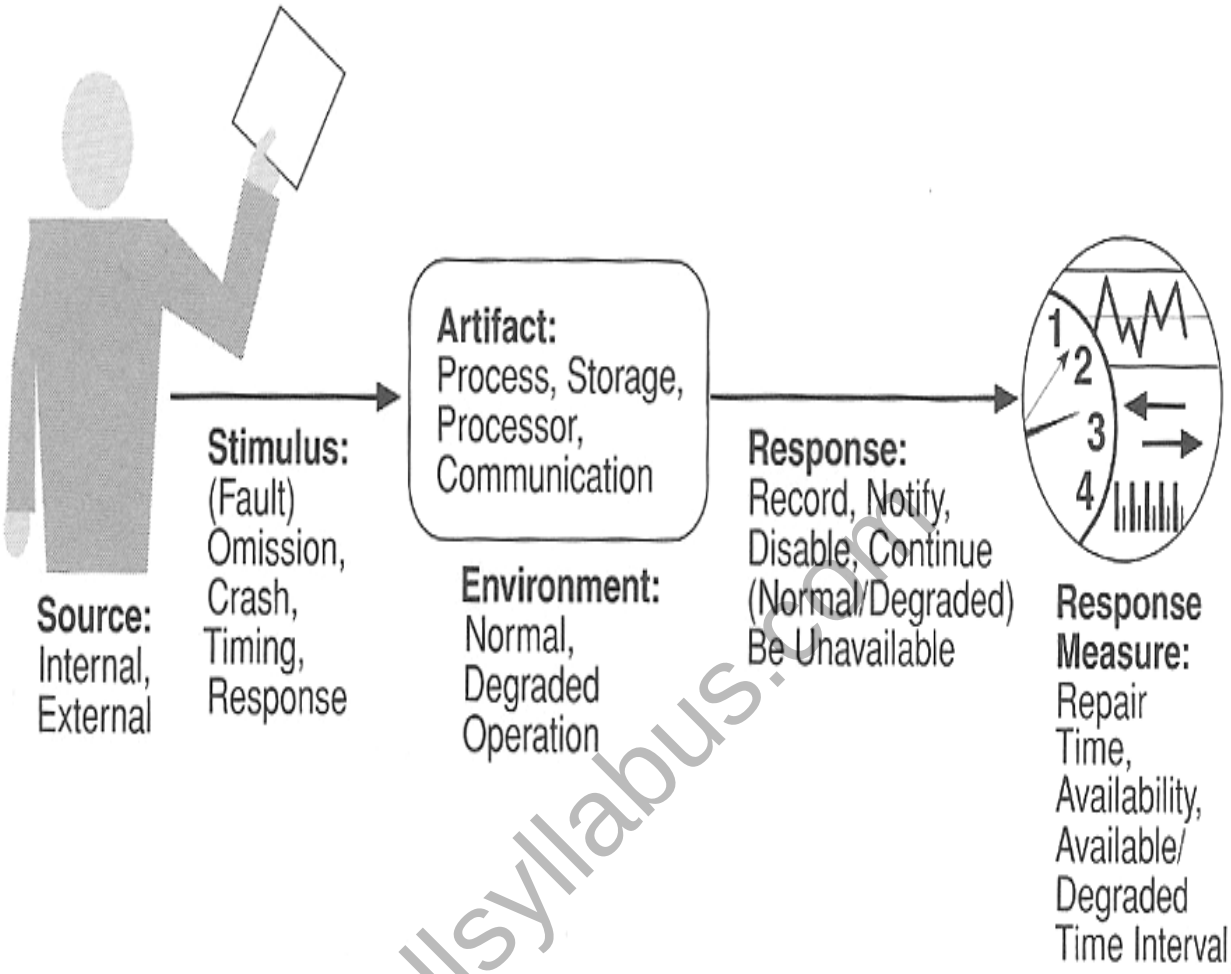
Quality of the system
Business quality
Quality of architecture itself

qualities discernable by observing the system execution

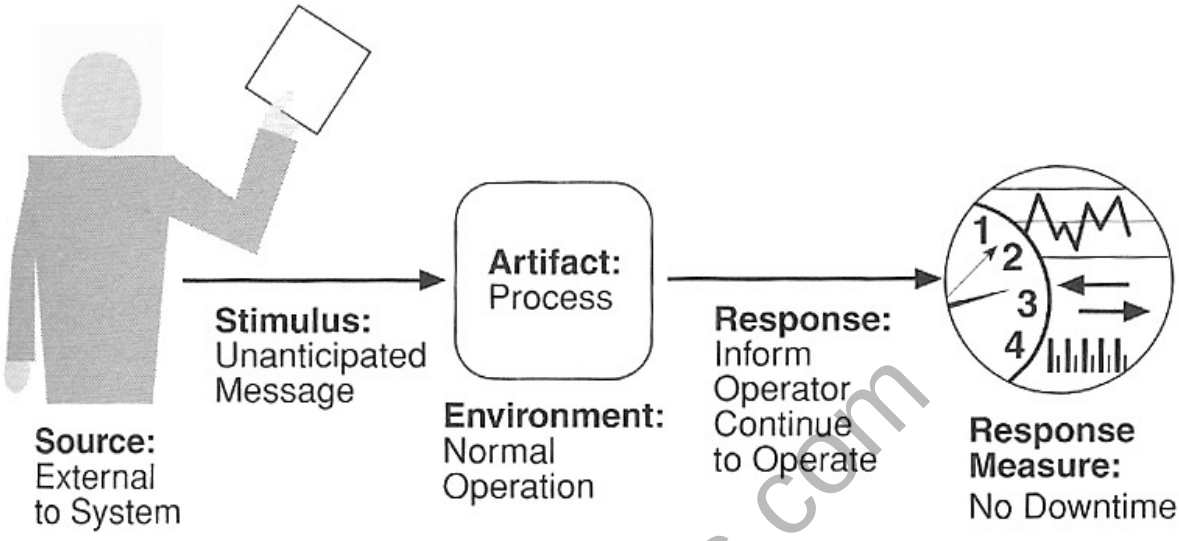
Availability
Performance
Security
Usability
functionality

qualities not discernable at run time:

Modifiability
Portability
Reusability
Integrability and testability



Groups of quality attribute discussion



All quality attributes are analyzed under the following groupings

Source, stimulus, artifact, environment, response and measure as indicated below

Availability

Availability deals with the following

1.Failures and faults

2. Mean time to failure, repair

3. Downtime

Source: internal, external

Stimulus: Faults: Omission, Crash, timing, response and measure as indicated below

Artifact: processors, channels, storage

Environment: normal, degraded

Response: logging, notification, switching to backup, restart, shutdown

Measure: availability, repair time, required uptime

Performance

Performance deals with the following

1.Event arrival patterns

periodic

Stochastic and or sporadic (in random and or recurring)

2.Event servicing

3.latency

4.jitter

5.Throughput

Systems performance can be simulated by a stochastic queuing model of the system

Analyze performance at the architectural level by:

Looking at the arrival rates and distributions of service requests

Processing times

Queuing sizes

Latency

Frequently compromised as the other qualities have emerged as important competitors

Source: external, internal

Stimulus: event arrival pattern

Artifact: system services

Environment: normal, overload

Response: change in mode?

Measure: latency, deadline, throughput, jitter, miss rate, data loss

Security

Security deals with the following

1. Attack or threat

2. Confidentiality

3. integrity

4. Assurance

5. Availability

Source: user/system, identified?

Stimulus: display info, change info, access services, deny services

Artifact: services, data

Environment: online/offline, connected?

Response: logging, block access, notification

Measure: time, probability of detection, recovery

Modifiability

It deals with changes taking place in the application

Source: developer, administrator, user

Stimulus: add/delete/modify function or quality

Artifact: UI, platform, environment

Environment: design, compile, build, run

Response: make change and test it

Measure: effort, time, cost

Testability

Testability deals with the following

1.Probability of fault discovery

2.Need to control components

3.Need to observe component failure

Source: developer, tester, user

Stimulus: milestone completed

Artifact: design, code component, system

Environment: design, development, compile, deployment, run

Response: can be controlled and observed

Measure: coverage, probability, time

Usability

Usability deals with the following

1.Learning

2.Using efficiently

3.Minimizing errors

4.Adapting to user needs

5.Increasing confidence and satisfaction

Source: end user

Stimulus: wish to learn/use/minimize errors/adapt/feel comfortable

Artifact: system

Environment: configuration or runtime

Response: provide ability or anticipate

Measure: task time, number of errors, user satisfaction

SUMMARY OE QUALITY ATTRIBUTES :

Availability- Unexpected events, non occurrence of expected events.

Modifiability- Request to add/delete/change/vary functionality, platform quality attributes or capacity

Performance-periodic, stochastic or sporadic

Security-tries to display ,modify, change/delete information, access or reduce availability to system services

Testability-Completion of phase of system development

Usability-Wants to learn system features, use a system efficiently, minimize the impact of errors, adapt the system, feel comfortable.

Business Qualities

Business Qualities deals with the following

1.Time to market

2.Cost and benefit

3.Projected lifetime

4.Target market

5.Rollout schedule

6.Integration with legacy

7.Architectural Qualities

Business Qualities

The business qualities are usually considerations of cost, schedule, market and marketing. Like the System Quality attributes they are also ambiguous. So, we need scenarios like we saw with system qualities, to guide in the design process.

The various business qualities we will look at are: Time to market, Cost and benefit, Projected lifetime of the system, Targeted market, Rollout schedule, and Integration with legacy systems.

Time to market becomes very important when there is severe competitive pressure or if the window of opportunity for introduction of system or product to the market is short. This would result in pressure to buy commercial off-the-shelf (COTS) products or reuse elements from previous projects. Good architecture can enable the ability to insert or deploy a subset of the system by using the right levels of abstraction in defining the elements of the system.

Balancing cost and benefit is something architecture has to do. For example, architecture relying on out-of-house technology will be more expensive. Similarly, highly flexible architecture will be more expensive to build than rigid architecture. But the benefit will be that it will be less costly to maintain and modify.

If a project has a long projected time, then the qualities of portability, scalability, modifiability becomes very important. To assure these, one has to build extra layers for these. This would increases time to market, but it will be worth it.

Targeted market will have a heavy influence on architecture. In the case of general purpose software, it will be the platform and feature set that will determine the size of potential market, and portability and functionality is key here. If the effort is to attract large market with collection of related products, then it would beneficial to design a product-line approach. In this the core system will be common, this will usually have to be designed portable to different platform (hardware or operating system). Around the core layers of specific software can be constructed.

Rollout schedule will need to be considered if the system is planned to be released with just the core functionality and more features will be released in different phases. So, flexibility and customizability becomes important.

Integration with legacy systems could be of high marketing importance. So, architecture has to define appropriate integration mechanisms, and the constraints imposed by these have to be analyzed.

Qualities of the Architecture

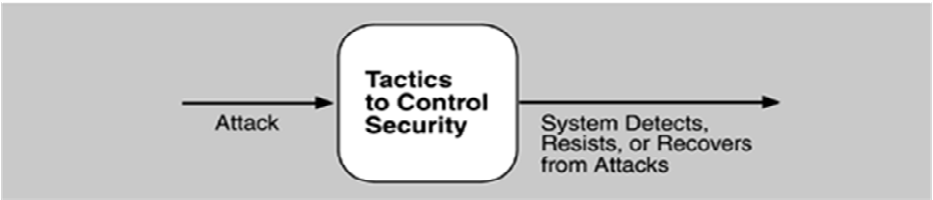
These qualities are related directly to the architecture itself. Conceptual integrity is an architectural quality that refers to the underlying theme or vision that unifies the design of the system at all levels.

Architecture should also ensure correctness and completeness. This is essential to confirm that all the functional requirements of the system are met, and quality attributes are achieved as planned.

Buildability is an architectural quality that allows a system to be completed by available teams in a timely manner and also ensures that the system is open to certain changes as development progresses. Architecture can help achieve buildability by paying attention to how the system is decomposed into modules, and limiting dependencies between the modules. The architect then has to assign these modules to appropriate development teams. Limiting dependencies between the modules limits the dependencies between the development teams, which will maximize development parallelism.

Security Tactics:

Tactics for achieving security can be divided into those concerned with resisting attacks, those concerned with detecting attacks, and those concerned with recovering from attacks.



Goals of Security Tactics

Resisting Attacks

- Tactics for achieving security can be divided into those concerned with resisting attacks, those concerned with detecting attacks, and those concerned with recovering from attacks.
- *Authenticate users.* Authentication is ensuring that a user or remote computer is actually who it purports to be.
  - Passwords, one-time passwords, digital certificates, and biometric identifications
- *Authorize users.* Authorization is ensuring that an authenticated user has the rights to access and modify either data or services.
  - Classes of users can be defined by user groups, by user roles, or by lists of individuals.
- *Maintain data confidentiality.* Data should be protected from unauthorized access.
  - encryption
- *Maintain integrity.* Data should be delivered as intended
  - checksums or hash results
- *Limit exposure.* Attacks typically depend on exploiting a single weakness to attack all data and services on a host.
  - The architect can design the allocation of services to hosts so that limited services are available on each host.
- *Limit access.* Firewalls restrict access based on message source or destination port.
  - One configuration used in this case is the so-called demilitarized zone (DMZ).

Detecting an Attack

- usually through an intrusion detection system
- work by comparing network traffic patterns to a database.
- Frequently, the packets must be filtered in order to make comparisons.

Recovering from Attacks

can be divided into those concerned with restoring state and those concerned with attacker identification

The tactics used in restoring the system or data to a correct state overlap with those used for availability

Special attention is paid to maintaining redundant copies of system administrative data such as passwords, access control lists, domain name services, and user profile data.

The tactic for identifying an attacker is to maintain ***an audit trail.***

An audit trail is a copy of each transaction applied to the data in the system together with identifying information.

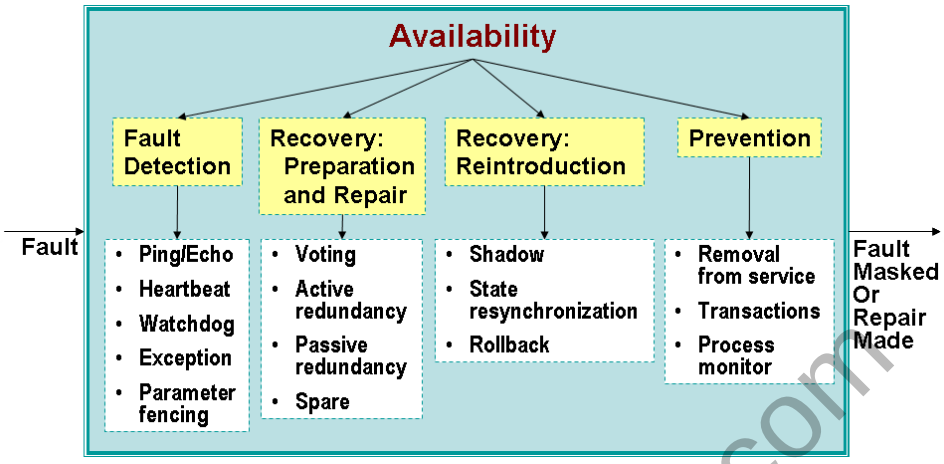
Achieving Qualities: Tactics

"An architectural tactic is a design decision that affects how well a software architecture addresses a particular quality attribute."

A system design is a collection of decisions. Some ensure achievement of the system functionality. A tactic is a design decision that influences the control of a quality attribute response.

Tactics for Availability

Availability is concerned with system failure and its consequences. The areas of concern here are: detection of system failure, frequency of failure, consequences of failure, how long a system is allowed to be out of operation



while handling failure, how failures can be prevented.

Availability: Fault Detection Tactics

Ping/Echo

One component issues a ping and expects to receive back an echo within a predefined time. If the echo is not received, it is assumed that the pinged component has failed and corrective action has to be initiated. Positives of this tactic are that there is limited interference of regular activities at the respondent side. An added bonus is that it determines not just reachability, but also the round-trip delay through the associated network path.

Heartbeat

In this tactic a subsystem emits a signal (the 'heartbeat') with a fixed frequency to a listener. If a signal is omitted, then the subsystem is assumed to have failed and corrective action will be initiated. The heartbeat signal can carry some data, too. So, the subsystem will send out data to be processed with the heartbeat message at pre-determined intervals.

Exceptions

When a fault is discovered the relevant subsystem would raise an exception. The faults can be divide-by-zero, bus and address faults, illegal program instructions.

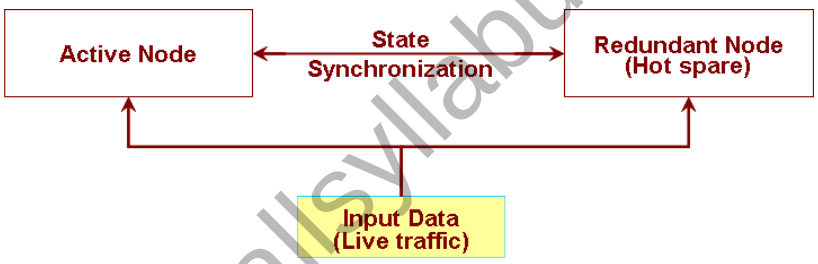
Availability: Tactics for Fault Recovery::Preparation and Repair

Voting

According to this tactic, a number of modules will compute the same function that is sent to a voter. If not all answers are identical, then a fault has occurred. The system may also take the majority of the answers as the right answer.

These different modules can be running on different hardware, could have been developed by separate implementation teams to achieve true redundancy. A typical implementation is having three versions of the modules. This system is known as Triple Modular Redundancy (TMR). Many spaceships and satellite systems often use TMR. Some ECC memory also uses this tactic.

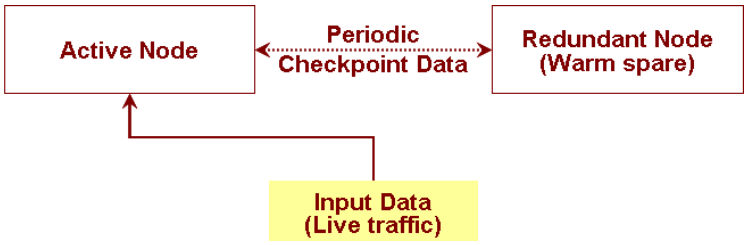
Active redundancy **(hot restart)**



All of the nodes (active or redundant spare) in a protection group receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain synchronous state with the active node(s). Recovery and repair can occur in time measured in milliseconds.

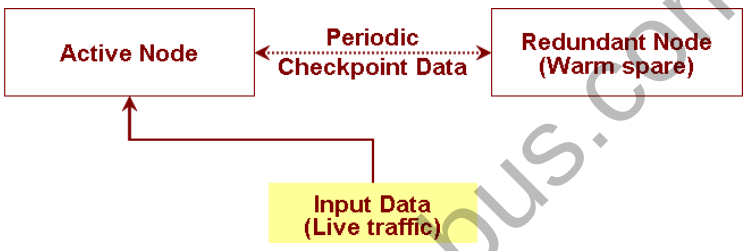
Passive redundancy **(warm restart)**

In this tactic only the active node process input traffic, with the redundant spare(s) receiving periodic state updates. Upon failure the system must ensure that the backup is sufficiently fresh before resuming services.



Spare

A standby spare computing platform is configured to replace many different failed components. The spare must be rebooted to the proper software configuration and have its state initialized when a failure occurs.



Availability: Tactics for Fault Recovery:: Reintroduction

Shadow operation

A previously failed component may be run in "shadow mode" for a short period of time to make sure it mimics the behavior of the working components before restoring it to service

State resynchronization

This is an example of a Refinement tactic. On reintroduction, components must have their states upgraded before returning them to service. As a refinement to the Active Redundancy tactic, the State Resynchronization occurs organically, as the active and standby components each receive and process identical inputs in parallel. As a refinement to the Passive Redundancy (warm sparing) tactic, State Resynchronization is based solely on periodic state information transmitted from the active components to the standby components.

Checkpoint/rollback

A checkpoint is the recording of a consistent state created either periodically or in response to specific events. When a fault occurs the system can be rolled back to that state.

Availability: Tactics for Fault Prevention

Removal from service

The removal of a component from service to undergo activities can sometimes mitigate potential system failures. An example might be taking a component of a system out of service and resetting the component in order to scrub latent faults such as memory leaks or fragmentation. This prevents accumulation of faults become service-affecting, resulting in system failure.

Transactions

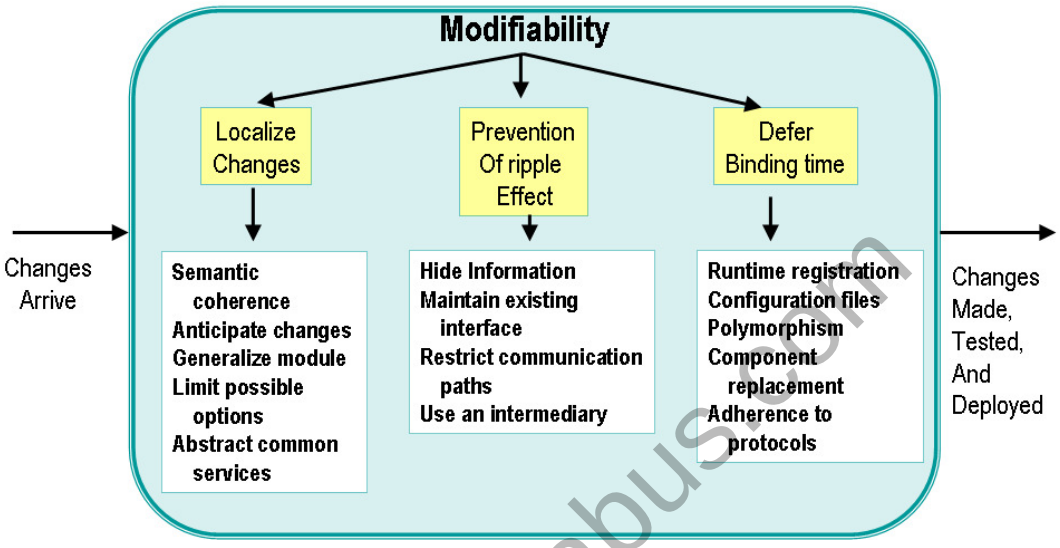
Transactions are the bundling of several sequential steps in which the entire bundle can be undone at once.

Process monitor

This tactic envisages a dedicated module that monitors for a fault in a process and deletes the nonperforming process and creating a new instance of it.

Modifiability Tactics

Maintenance generally presents the major cost factor of the lifecycle of software systems. Maintainability is the capability of the software product to be modified. The modifications may include corrections, improvements or adaptations of the software to changes in environment, and in requirements and functional specification. The ease with which a software system can accommodate changes is controlled by Modifiability tactics.



Modifiability tactics: Localize changes

Maintain semantic coherence

This refers to the relationships among responsibilities in a module. This tactic makes sure that all the responsibilities in a module are related and work together without excessive reliance on other modules. A related sub-tactic is to abstract the common services. This is relevant because modifications to a module that provide common services will need to be made only once rather than in each module where the services are used.

Anticipate expected changes

Considering the set of envisioned changes when doing decomposition facilitated the evaluation of assignment of responsibilities to the subsystems, with the aim to minimize the effects of the changes. This tactic is especially difficult in practice because it is nearly impossible to anticipate all possible changes that can be made during the lifetime of a system.

Generalize the module

The goal of this tactic is for a module to compute a broader range of functions based on input. The more general a module, the more likely that requested changes can be made by adjusting the input language rather than by modifying the module.

Restrict changes within reason

This tactic applies specially to product-line architectures, where variation can happen along many different dimensions, so much so that it becomes impractical. So, the architect has to use good judgment and restrict changes within reason. For example, a variation point in a product line may be allowing for a change of processor. In this case, a tactic might be to restrict processor changes to members of the same family.

Modifiability tactics: Prevent Ripple Effects

Hide information

Decomposition of the responsibilities for an entity into smaller pieces and choosing which information to make private and which to make public is very critical in preventing ripple effects. The tactic '*anticipate expected changes*' is related to this, because the anticipated changes will have to be encapsulated.

Maintain existing interfaces

If other modules depend on the name and signature of an interface of a module, maintaining this interface and its syntax allows other modules to remain unchanged. Patterns to implement this tactic include adding interfaces or adapter to modules. When a module provides some new service after a modification, add new interfaces to expose these services while maintaining the old interfaces. Another pattern when other modules are dependent only on the signature of the changed module is providing a stub.

This tactic is going to work only if there is a syntactical relationship. If there is a semantic dependency, quality of data or service dependency, resource usage/ or ownership dependency, these break down.

Restrict communication paths

In any system if there are multiple data producers or multiple data consumers, the dependencies will cause ripples when modifications have to be made. This tactic recommends restrict the modules with which a given module shares data.

Use an intermediary

This tactic recommends us to insert an intermediary between dependent modules that manages activities associated with the dependency. For example, if the location of a module needs to be changed, the intermediary used will be a name server, so that the run-time location of the module can be changed dynamically

Modifiability tactics: Defer Binding Time

This tactic allows change in time to deploy, and also allows non-developers to make changes to the system. Deferring binding time will of course come at the cost of requiring additional infrastructure to support the late binding.

The tactic of Runtime registration supports plug-and-play operation, but at an additional overhead to manage the registration. A more constrained variant of this tactic is to do these publish/subscribe registration at load time instead of at runtime.

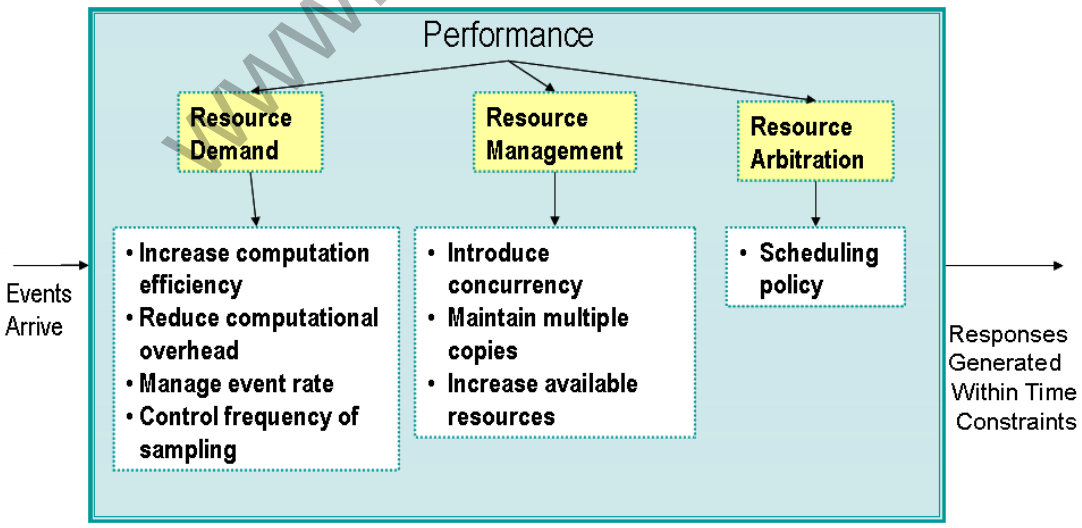
A frequently used method is to use Configuration files, which will set parameters at startup that will change the behavior of the system. Polymorphism allows late binding of method calls. Component replacement allows load time binding. To allow runtime binding of independent processes, adherence to defined protocols serves as the right tactic.

Performance tactics

The goal of performance tactics is to generate a response to an arriving event within some time constraint. The arriving event can be a single event or a stream of events. The event can be a message, expiration of a time interval, detection of a state change, etc. Performance tactics control the time within which a response is generated. Latency is the time between the arrival of an event and the generation of a response.

There are two basic contributors to latency. The first contributor is resource consumption. The resources include CPU, persistent data-stores, network communication bandwidth, and random access memory. Events go through a processing sequence which contributes to the overall latency of the response. The second contributor is blocked time. This could be due to contention for resources by competing subsystems. Another possibility of downtime, even when there is no contention when resources go offline or there is some failure in the system that causes the resources to become unavailable. Another cause for latency could be due to dependency on other computation by other modules.

Performance tactics come in three different categories. First category deals with resource demand whose aim is to reduce resources required to process an event stream. The second category deals with resource management, whose effort is to reduce the number of events processed. The last category entails resource arbitration, which strives to control use of resources.



Performance Tactics: Resource Demand

The following two tactics attempts to reduce resources required to process an event stream. Here are the tactics.

Increase computational efficiency

This can be done by improving algorithm efficiency at critical places or trade one resource for another.

Reduce computational overhead

An example of this tactic is to eliminate intermediaries, which will reduce latency. Note the conflict between modifiability and performance.

The following two tactics attempts to reduce the number of events processed.

Manage event rate

If the system was over-engineered, then it might be possible to reduce event rate. An example might be to reduce the frequency at which environmental variables are monitored.

Control frequency of sampling

An example of this is to sample queued requests at lower frequency, which will result in less resources consumed, but might cause some requests to be lost.

The following two tactics attempts to control use of resources

Bound execution times

This is not a very general tactic. There could some special cases where it might be possible to limit how much execution time is used to respond to an event.

Bound queue lengths

Like the previous tactic under some special circumstances it might be possible to control the maximum number of queue arrivals to save on resources.

Performance Tactics: Resource Management

If the demand for resources isn't controllable, they might be managed by these tactics to manage resources more efficiently

Introduce concurrency

Blocked time can be reduced if requests can be processed in parallel.

Maintain multiple copies of either data or computations

Caching is a tactic in which data is replicated that will result in less contention on resources. But this tactic will necessitate tactics to perform synchronization of copies.

Increase available resources

Add additional or faster processors, memory, or faster networks can reduce latency. Of course, the cost of increasing resources has to be offset by the benefits they provide.

Performance Tactics: Resource Arbitration

When there is contention for a resource, the resource must be scheduled. Competing criteria for scheduling include optimal resource usage, request importance, minimizing the number of resources used, minimizing latency, maximizing throughput, and preventing starvation to ensure fairness.

We will look at some common scheduling strategies

1. **FIFO**: First-in/First-out is a tactic that can be used if all requests are equal

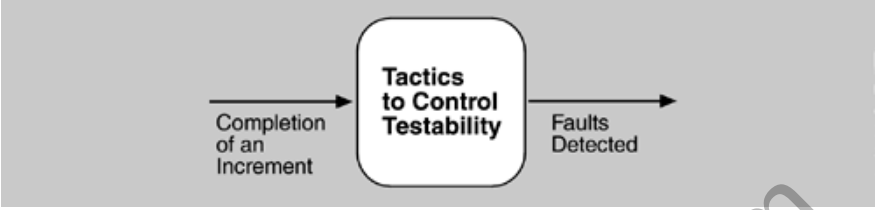
2. **Fixed-priority scheduling**: Fixed-priority scheduling assigns each source of resource requests a particular priority and assigns the resources in that priority order. The priority might be based on *semantic importance*, where priorities are assigned to request categories statically. Another way to do this is called *deadline monotonic*. Under this policy tasks with shorter deadlines are assigned higher priori. *Rate monotonic* policy is another variant where tasks with shorter periods are assigned higher priority.

3. **Dynamic priority scheduling**: There are a couple of ways to perform dynamic priority scheduling. The round robin is a fair scheduling method. Another way to do scheduling is to give process closest to its deadline is highest priority (*earliest deadline first*).

4. **Static scheduling**: In this schedule with pre-emption points and sequence are determined offline.

Testability Tactics

The goal of tactics for testability is to allow for easier testing when an increment of software development is completed.



- since testing consumes such a high percentage of system development cost, anything the architect can do to reduce this cost will yield a significant benefit.
- Executing the test procedures requires some software to provide input to the software being tested and to capture the output. This is called a test harness.
- We discuss two categories of tactics for testing:
  - providing input and capturing output, and internal monitoring.

Input/Output

- There are three tactics for managing input and output for testing.
  - *Record/playback.* Record/playback refers to both capturing information crossing an interface and using it as input into the test harness.
  - information crossing an interface during normal operation is saved in some repository and represents output from one component and input to another.
  - *Separate interface from implementation*. Separating the interface from the implementation allows substitution of implementations for various testing purposes.
  - Stubbing implementations allows the remainder of the system to be tested in the absence of the component being stubbed.
  - *Specialize access routes/interfaces.* Having specialized testing interfaces allows the capturing or specification of variable values for a component through a test harness as well as independently from its normal execution.

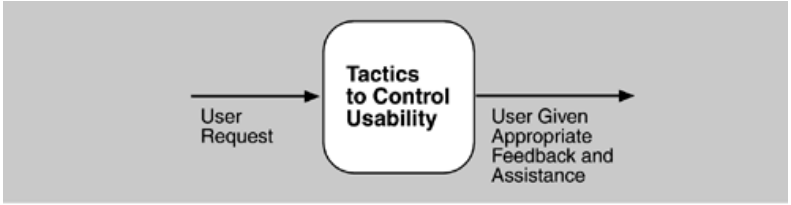
Internal Monitoring

- Built-in monitors. The component can maintain state, performance load, capacity, security, or other information accessible through an interface.
- Can be permanent interface to components or introduced temporarily
- common technique is to record events when monitoring states have been activated.

Usability Tactics:

Two types of tactics support usability

- runtime, includes those that support the user during system execution.
- iterative nature of user interface design that supports the interface developer at design time.



Runtime Tactics

- During execution, the usability is enhanced by
  - the feedback to user about the system
  - Allowing user to issue commands: cancel, undo, aggr
- According to Researchers in Human – computer interaction:
  - User Initiative
  - System Initiatives
  - Mixed Initiatives
- Our Usability scenario always combines first two to give last type of interaction
- *User Initiative* – the architect designs response similar to other functionality.
- Must enumerate the responsibilities of the system in response
- The user issues a cancel command
  - System must be listening to it
  - Command should be killed
  - Resources should be freed
  - Collaborating components must be informed to take necessary action
- *System Initiative* – rely upon a ***model***
  - About the user
  - Task being undertaken by the user, or
  - System state itself
- The system initiative tactics are those that identify the models the system uses to predict either its own behavior or the user's intention.
- this information will enable an architect to more easily tailor and modify those models.
- Tailoring and modification can be either dynamically based on past user behavior or offline during development.
- *Maintain a model of the task.* In this case, the model maintained is that of the task.
  - The task model is used to determine context
- *Maintain a model of the user.* In this case, the model maintained is of the user.
  - It determines the user's knowledge of the system, the user's behavior in terms of expected response time,
- *Maintain a model of the system.* In this case, the model maintained is that of the system.
  - It determines the expected system behavior

Design-time Tactics

- User interfaces are typically revised frequently during the testing process.
- the usability engineer will give the developers revisions to the current user interface design and the developers will implement them.
- This leads to a tactic that is a refinement of the modifiability tactic of semantic coherence
- Separate the user interface from the rest of the application.
    - Localizing expected changes is the rationale for semantic coherence.
    - the user interface is expected to change frequently both during the development and after deployment
    - The software architecture patterns developed to implement this tactic and to support the modification of the user interface are:
        - Model-View-Controller
        - Presentation-Abstraction-Control
        - Seeheim
        - Arch/Slinky

Relationship of Tactics to Architectural Patterns

An architect usually chooses a pattern or a collection of patterns designed to realize one or more tactics. However, each pattern implements multiple tactics, whether desired or not.

Example: Active Object design pattern: The Active Object design pattern decouples method execution from method invocation to enhance concurrency and simplify synchronized access to objects that reside in their own thread of control.

- The pattern consists of six elements:
    - a proxy, which provides an interface that allows clients to invoke publicly accessible methods on an active object
    - a method request, which defines an interface for executing the methods of an active object
    - an activation list, which maintains a buffer of pending method requests
    - scheduler, which decides what method requests to execute next
    - a servant, which defines the behavior and state modeled as an active object
    - future, which allows the client to obtain the result of the method invocation

The motivation for this pattern is to enhance concurrency—a performance goal. Thus, its main purpose is to implement the "introduce concurrency" performance tactic.

There are other tactics this pattern involves, however.

- Information hiding (modifiability). Each element chooses the responsibilities it will achieve and hides their achievement behind an interface.
- Intermediary (modifiability). The proxy acts as an intermediary that will buffer changes to the method invocation
- Scheduling policy (performance). The scheduler implements some scheduling policy.

The analysis process involves understanding all of the tactics, and the design process involves making a judicious choice

UNIT 4

Layers Architectural Pattern

In the layer architecture, Application is decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

Context

The context in which one might consider this pattern is a large system that requires decomposition. One way to decompose a system is to segment it into collaborating objects. In large systems a first-cut rough model might produce hundreds or thousands of potential objects. Additional refactoring typically leads to object groupings that provide related types of services. When these groups are properly segmented, and their interfaces consolidated, the result is a layered architecture.

Problem Description

The dominant characteristic is a mix of high and low level issues, where high level operations rely on lower level ones. There is also necessity for bidirectional communication flows. Portability between platforms is desirable. The external boundaries of the system are specified, usually in terms of functional Interfaces that system must adhere to. Also because of the complexity of the higher level tasks, mapping of high-level tasks onto platform is not straightforward.

Forces to Balance

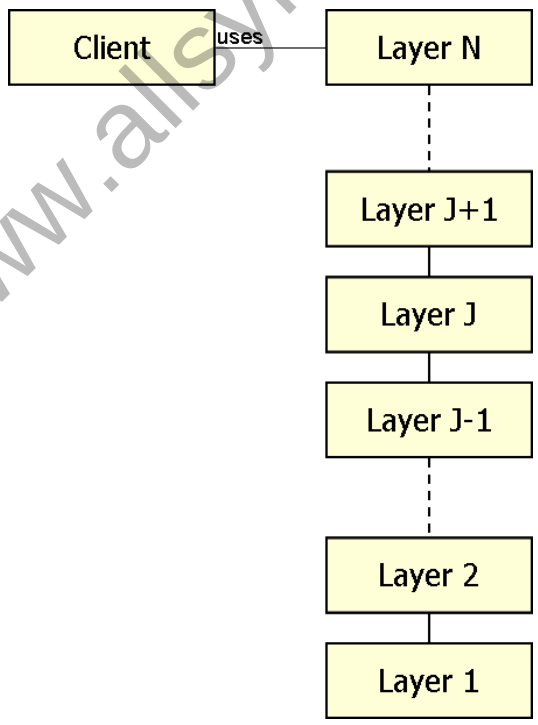
There are always several forces to balance for the architect in this context. A force to balance is that late changes to subsystems should not “ripple” through the system. The interfaces should be stable, and probably some standards have to be followed. Another force to balance is that some subsystems should be exchangeable. There might also be the need to allow alternative implementations. The possibility of platform change (Operating System or Hardware) might be there. Some systems might require run-time changes by reconfiguration. Also, it may be necessary to build other systems that have same low-level issues.

While decomposing the system into layers, similar responsibilities should be grouped. At the same time each component should be coherent. If one component implements divergent issues, its integrity is lost. So, an important force to balance is to resolve this potential conflict between grouping and coherence. In many domains, there will be no 'standard' component granularity. Components will have to undergo iterative decomposition till we have simple components. Another important force to balance when deciding on the number of layers is that crossing component boundaries degrades performance. From the development task point of view, the architect will have to subdivide the work among a team.

Solution

The solution is to structure the system as a series of layers. We start at the lowest level (layer 1). Then, work our way up the abstraction ladder, putting layer J on layer J-1 until we reach the top layer, which the user of the system will interact with usually.

We have to ensure that most services provided by layer J are composed of services from layer J-1. Of course layer J's services may depend on other services within Layer J.



Dynamics

Scenario I (top-down requests)

In this scenario, client or user issues request to layer N. Layer N calls layer N-1 for supporting subtasks. Layer N-1 provides these, and also calls layer N-2, and so on until layer 1 is reached. Lowest level services are performed. If necessary, replies are passed back up the chain.

Note that the layer J is at a higher level of abstraction than Layer J-1. Map high-level service onto more primitive ones. So, for each request received from above, typically several requests are made to layers below.

Scenario II (only partial flow down)

Sometimes, a layer may be able to service the request, without making requests lower down. An example is when you have a caching layer, requests sometimes need not be sent down to layer 1. The issue in this case is the need in these caching layers to maintain state information.

Scenario III (bottom-up notifications)

In this scenario, a chain of events start at the bottom most layer. An example might be in the case of a device driver in layer 1 detects input. The device driver translates this into internal format, and notifies layer 2, which starts interpreting it, and notifies layer 3. Process continues up to highest layer.

Variation in this scenario is when incoming notifications may be condensed into a bulk notification.

Scenario IV (only partial flow up)

This is when a layer may be able to handle a notification without forwarding it.

Scenario V (communications)

This scenario arises when two stacks of N layers. The layers are typically known as “protocol stacks”. Here the top layer on one stack issues a request to send data. Subsequent layers perform their services and pass on the data, until it hits layer 1. Data is transmitted, and the reverse procedure occurs on the receiving side.

Implementation

1. Define the abstraction criterion

The criteria used might be the conceptual distance from platform or the degree of conceptual complexity. But usually it is a mix of criteria, such as distance from hardware at lower levels, and conceptual complexity at higher levels

2. Determine the number of abstraction levels

Typically each abstraction level corresponds to one layer. But not in all cases would this mapping from levels to layers be obvious. At this stage, we should also think about tradeoffs. Too many layers may impose unnecessary overhead, and too few layers can result in poor structure.

3. Name layers, and assign tasks to them

If we take the top-down approach, then task of the highest layer is the overall system task, as perceived by the client. All other layers are helpers to higher layers. If we take the bottom-up approach, we need to think of lower layers as providing an infrastructure on which higher layers can build. Designing from bottom up requires experience and foresight to find the right abstractions for the lower layers, so that they will adequately support the higher layers.

4. Specify the services

Layers must be strictly separated; that is, no component may be spread over several layers. Generally locate more services in higher layers and fewer services in lower ones. This prevents developers from having to learn a large number of similar primitives.

5. Refine the layering

We have to iterate steps 1-4 a few times. This is because usually it is not possible to define abstraction criterion without thinking of layers and services. Similarly, it is usually wrong to define components and services first, and then impose layered structure. The natural solution is to iterate the first four steps to achieve good layering.

6. Specify an interface for each layer

If we take the Black-box approach, then layer J should have a "flat interface" that offers all its services to Layer J+1. If we take the white-box approach, then layer J+1 sees the internals of Layer J.

The recommended approach is black-box approach because it supports system evolution better than other approaches. Exceptions can be made for efficiency, but these are very seldom required or truly appropriate.

7. Structure individual layers

When an individual layer is complex, it should be broken into well-defined components

8. Specify communication between layers

Most often used method of communication is the push model, where the required information passed by layer J+1 to layer J in the service request/method call. Alternative method is to use the pull model, where lower layer requests information from upper once it has been invoked. The disadvantage is that it introduces additional dependencies between layers, but it can be solved by using call-backs.

9. Decouple adjacent layers

Systems that use top-down invocation sequences use top down coupling, where upper layers know about, and call, lower layers, and 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 call-backs can be used to preserve the one-way coupling. This is most effective when a limited event vocabulary is used in the call-backs. Where lots of communication is bottom up, we may need to have two-way coupling.

One design principle to remember at this stage is to code against interfaces rather than implementations.

10. Design error handling strategy

Error handling is often problematic in layered systems due processing time and programming effort. An error handled in layer where it occurred, or forwarded to the next higher layer. If it is forwarded then the error information must be transformed into something meaningful. The recommended principle is to try to handle errors at lowest possible level.

Variants

Relaxed layered system

In this variant, each layer may use services of all layers below, not just adjacent layers. You might have some layers that are partially opaque. The major gain is in flexibility, performance. But the price is paid in loss of maintainability.

This is generally found in infrastructure systems (Operating Systems or frameworks) which generally require little maintenance but requires high performance.

Layering through inheritance

This variant found in OO systems. Here lower layers are base classes, and higher layers inherit from lower layers. S, this allows for selective modification of lower layers. But the major problem is that this variant closely ties layers together, which has severe repercussions in modifiability.

Consequences

Advantages

Reuse of layers: If abstractions and interfaces are well-defined reusability is enhanced. Black box reuse dramatically reduces development effort and number of defects

Support for Standardization: Clearly defined and commonly accepted levels of abstraction enable standardization. This allows for the use third-party libraries and frameworks.

Dependencies are kept local: Layer pattern generally minimizes ripple, as changes are confined to one layer as long as interfaces are not changed. This also supports portability

(by replacing lowest-level components). The limitation in the range of interactions between components simplifies correctness reasoning and also enhances testability

Exchangeability: Different implementations of a layer can be exchanged as long as interfaces are the same.

Disadvantages

Cascades of changing behavior: If the behavior of a layer changes dramatically (e.g., replace 10 MBPS Ethernet layer with 155 MBPS ATM) this may have large ripple effect through the architecture.

Lower efficiency: System performance may suffer from unnecessary layering overhead, as multiple transformations are performed on data as it passes through each layer.

Difficulty in establishing correct granularity of layers: It is not always easy to structure in clean layers as it may be difficult to find the right levels of abstraction.

Pipes and Filters Architectural Patterns

Like the *Layers* pattern, Pipes and Filters pattern allows for a systematic decomposition of a system that needs to process a stream of data. The solution is split into several sequential steps. The filter component implements one step of the process. Pipes carry data between adjacent filters.

Forces to Balance

There are several forces to balance when designing a system using this pattern. The processing of data elements can be broken down into a sequence of individual transformations. Small processing steps are easier to reuse in different contexts than larger components. But this might impact performance. It is also important to ensure that non-adjacent processing steps do not share information. Another aspect that will need to be accommodated is that different sources of input data might exist. We also need to be able to present or store final results in various ways. We need to avoid explicit storage of intermediate results in files as this will clutter directories and is error-prone. For the sake of improving performance we may not want to rule out multi-processing steps probably to run some filters concurrently. We should also be able to enhance the system by exchanging processing steps and recombining steps.

Solution

The Pipes and Filters pattern divides the task at hand into several sequential processing steps, each of which is connected by a data flow. This is done in such a way that the output of a step is input to the next processing step. Each of these processing steps is implemented by a filter component. A filter consumes and delivers data incrementally to achieve low latency and enable real parallel processing. Input to the system is a data source, such as a file. The output of the system flows into a data sink, such as a file or display device. Pipes connect the input, all the filters in the system, and finally the output. This sequence of filters combined by pipes is called a processing pipeline.

Structure

Filters

The filters are the processing units of the pipeline. Some filters might be performing some conversion task, such as converting Extended Binary Coded Decimal Interchange Code (EBCDIC) to ASCII. Other filters might be involved in providing some enrichment service, such as adding information to incoming messages. Still others might be performing filtering, such as discarding messages that match a specific criterion. There could be filters that does batching work, such as aggregating multiple incoming messages and sending them together in a single outgoing message. Some filters might get inputs from multiple filters and they may perform consolidation, such as combining the data elements of multiple related messages into a single outgoing message to the downstream filter.

Filters should use little local context in processing streams and it should not preserve state information between instantiations.

Filters can be classified as active or passive. An active filter has a control loop that runs in its own process or thread, and perpetually reads data from its in-pipe, processes it, then writes it to its out-pipe. If there is no data in the in-pipe, then the active filter enters a wait state. A passive filter, when activated, reads a single message from its in pipe, processes it, and then writes the result to its out pipe. There are two types of passive filters. A data driven filter (also called the push variant) is activated when another filter writes data into its in pipe. A demand driven filter (also called the pull variant) is activated when another filter attempts to read from its empty out pipe.

Pipes

A pipe transfers data from a data source to a data sink. Pipes can be between:

two threads of a single process (e.g., Java IO Streams) where the stream may contain references to shared language objects. Pipes can also be between two processes on a single host computer (e.g., Unix Named Pipes). Here the stream may contain references to shared operating system objects (e.g., files). In some systems, pipes can be between two processes

in a distributed system (e.g., Internet Sockets) where the stream contents normally limited to “raw bytes”.

Data Source

The data source provides input to the system, as a sequence or stream of data. This can be by actively pushing the data values to the first filter, or be passive and let the first filter pull data.

Data Sink

The results from the last filter in the system provides the final output to the Data Sink. Just like the Data Source, the Data Sink can be active or passive. Either the last filter can push data onto the Data Sink or the Data sink can actively pull results out of the last filter.

Implementation

The first step in implementation will be to divide the functionality of the problem into a sequence of processing steps. We have to define the type and format of the data to be passed along each pipe. It might be better to have some uniform format for all the filters. This is recommended if high flexibility is desired, although it might be inefficient, because we will need some filter components to make the data transformations.

The next step is to determine how to implement each pipe connection. we can do this either as direct call or utilize a separate pipe mechanism. Passive filters can be implemented as functions to provide pull activation. If push activation is needed, then the filters can be implemented as procedures. Active filters can be implemented as processes or threads in the pipeline program. The next step is to design error-handling in the system. Decisions have to be taken about how to handle the errors, and how to recover from the errors. The last step is to configure the pipes-and-filters system and initiate the processing.

Benefits

Reuse of filter elements is a major benefit. Filters that implement simple transformations typically encapsulate fewer assumptions about the problem they are solving than filters that implement complex transformations. The ease of filter recombination encourages filter reuse.

Improved performance is achieved because a Pipes and Filters solution processes messages as soon as they are received, thereby reducing latency.

Efficiency can be achieved by parallel processing. Since active filters run in separate processes or threads, pipes-and-filters systems can take advantage of a multiprocessor

Intermediate files is not necessary, but during debugging phase this might be necessary, which this pattern accommodates.

Flexibility is attained by filter exchange as it is easy to exchange one filter element for another with the same interfaces and functionality.

Also, modifiability is improved as we can change the filter configuration dynamically.

We also get flexibility by reconfiguring a pipeline to include new filters or perhaps to use the same filters in a different sequence.

Flexibility of exchange and recombination and ease of reuse enables the rapid creation of prototype systems

Liabilities

Designing filters typically requires expert domain knowledge. It also requires several good examples to generalize from. The challenge of identifying reusable transformations makes filter development an even more difficult endeavor. Designing the system will be complex.

Assessing the state of the system is complex. The Pipes and Filters pattern distributes the state of the computation across several components. This distribution makes querying the state a complex operation. We can over come this by sharing state information but it is expensive or inflexible, because each filter should have the information encoded, transmitted, and at the receiver's end it should be then decoded.

Efficiency gain by parallel processing can sometimes be an illusion due to several reasons. Sometimes, the costs of data transfer can be high, compared to cost of computation by a monolithic component. Also, when filters operate at different speeds, synchronization of filters via pipes may stop and start filters often, especially when a pipe has only a small buffer. Also context-switching in a single processor machine is expensive. In some systems, there could be just one non-incremental filter, such as the Unix sort, that can become the bottleneck of a system.

There is also the data transformation overhead. The use of a single data channel between filters often means that lot of transformation of data must occur, e.g., translation of numbers between binary and character formats

Error handling is especially difficult because filters have no knowledge of the context that they operate in. It is often difficult to detect errors in pipes-and-filters systems. Also, recovering from errors is even more difficult, so often the whole system might have to be restarted.

Maintainability effort is also increased because a Pipes and Filters configuration usually has more components than a monolithic implementation. Each component adds maintenance effort, system management effort, and opportunities for failure.

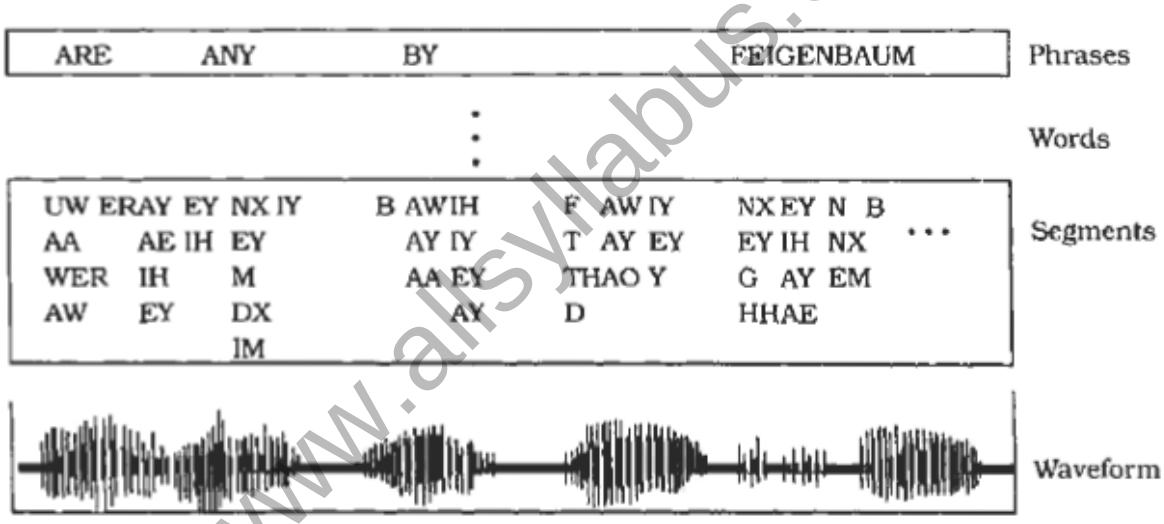
Blackboard Architectural Pattern

Definition:

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.
- The desired output is a machine representation of the corresponding English phrases.
- The transformations involved require acoustic- phonetic, linguistic and statistical expertise
- For example. one procedure divides the waveform into segments, another procedure checks the syntax of candidate phrases



The input is the waveform at the bottom, and the output consists of the phrase 'are any by Feigenbaum'.

Context:

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

Problem:

- The Blackboard pattern tackles problems that do not have a feasible deterministic solution for the transformation of raw data into high-level data structures
- *Example domains:* Vision, image recognition, speech recognition and surveillance

- Problem is, when decomposed, they span into several domains
- The solutions to the partial problems require different representations and paradigms
- In many cases no predetermined strategy exists for how the 'partial problem solvers' should combine their knowledge.
- You may also have to work with uncertain or approximate knowledge.
- Each transformation step can also generate several alternative solutions.
- It is often enough to find an optimal solution for most cases, and a suboptimal solution, or no solution, for the rest.

The limitations of a Blackboard system therefore have to be documented carefully and results should be verified.

Artificial Intelligence systems have been used with some success for such problems.

This type of expert system structure is inadequate for a speech recognition system for 3 reasons:

- All partial problems are solved using the same knowledge representation. But components of speech recognition process differ widely.
- The expert system structure provides only one inference engine to control the application of knowledge. Different partial problems with different representations require separate inference engines.
- In a 'classical' expert system, control is implicit in the structure of the knowledge base.

Solution:

- The idea behind the Blackboard architecture is a collection of independent programs that work cooperatively on a common data structure.
- Each program is specialized for solving a particular part of the overall task.
- These specialized programs are independent of each other.
- They do not call each other, nor is there a predetermined sequence for their activation.
- 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 the specialized programs. This data-directed control regime is referred to as *opportunistic problem solving*.
- During the problem-solving process the system works with partial solutions that are combined, changed or rejected.
- The set of all possible solutions is called the *solution space*, and is organized into levels of abstraction.
    - *Lowest level:* Internal representation of input
    - *Highest level:* potential solution of the task

Structure:

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

1. BLACKBOARD:

- The blackboard is the central data store.
- Elements of the solution space and control data are stored here.
- *Vocabulary* - the set of all data elements that can appear on the blackboard.
- The blackboard provides an interface that enables all knowledge sources to read from and write to it.
- All elements of the solution space can appear on the blackboard.
- *hypothesis or blackboard entry* - solutions that are constructed during the problem solving process and put on the blackboard
- A hypothesis usually has several attributes such as *abstraction level*, that is, its conceptual distance from the input.

In our example:

- The solution space for the speech recognition example consists of acoustic-phonetic and linguistic speech fragments.
- The levels of abstraction are signal parameters, acoustic-phonetic segments, phones, syllables, words, and phrases.
- The blackboard can be viewed as a three-dimensional problem space with the time line for speech on the X-axis, increasing levels of abstraction on the Y-axis and alternative solutions on the Z-axis.

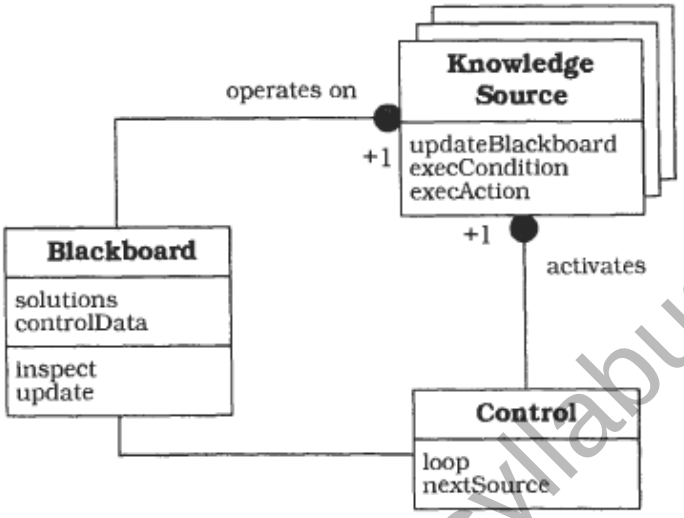
2. Knowledge Sources

- Knowledge sources are separate, independent subsystems that solve specific aspects of the overall problem.
- None of them can solve the task of the system alone - a solution can only be built by integrating the results of several knowledge sources.
- In our example, we specify solutions for the following partial problems: defining acoustic-phonetic segments, and creating phones, syllables, words and phrases.
- For each of these partial problem we define one or several knowledge sources.
- Knowledge sources do not communicate directly.
- They therefore have to understand the vocabulary of the blackboard.
- Often a knowledge source operates on two levels of abstraction – forward and backward reasoning
- Knowledge sources are split into a ***condition-part*** and ***an action-part.***
    - The condition part evaluates the current state of the solution process.
    - The action-part produces a result that may cause a change to the blackboard's contents.

3. Control Component:

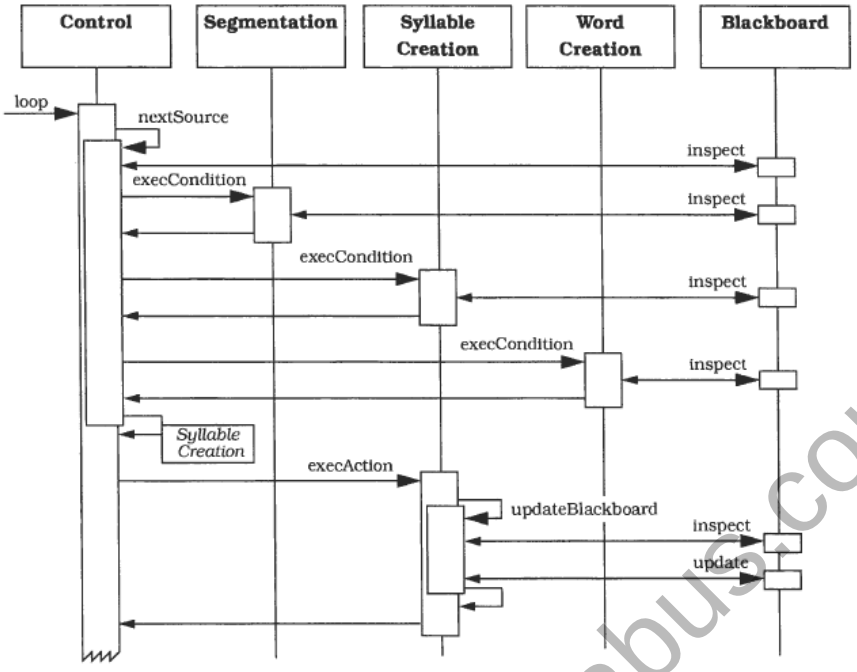
- The control component runs a loop that monitors the changes on the blackboard and decides what action to take next.
- It schedules knowledge source evaluations and activations according to a knowledge application ***strategy.***
- The strategy may rely on ***control knowledge sources.***
- These special knowledge sources do not contribute directly to solutions on the blackboard, but perform calculations on which control decisions are made.

Relationship between three components:



Courtesy: Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sornmerlad, Michael Stal, *Pattern Oriented Architecture: A system of Patterns*, Volume 1, Wiley Publications

Dynamics:



Implementation

Step1: *Define the Problem*

- Specify the domain of the problem and the general fields of knowledge necessary to find a solution.
- Scrutinize the input to the system. Determine any special properties of the input such as noise content or variations on a theme
- Define the output of the system.
- Detail how the user interacts with the system.

Step2: *Define the solution space for the problem*

- We distinguish *intermediate* and *top-level solutions* on one hand, and *partial* and *complete solutions* on the other.
- So perform following steps:
  - Specify exactly what constitutes a top-level solution.
  - List the different abstraction levels of solutions.
  - Organize solutions into one or more abstraction hierarchies.

Step3: *Divide the solution process into steps:*

- Define how solutions are transformed into higher-level solutions.
- Describe how to predict hypotheses at the same abstraction level.
- Detail how to verify predicted hypotheses by finding support for them in other levels.
- Specify the kind of knowledge that can be used to exclude parts of the solution space.

Step4: *Divide the knowledge into specialized knowledge sources with certain subtasks.*

- These subtasks often correspond to areas of specialization.
- There may be some subtasks for which the system defers to human specialists for decisions about dubious cases, or even to replace a missing knowledge source.
- Examples of knowledge sources are segmentation, phone creation, syllable creation, word creation, phrase creation, word prediction and word verification.

Step5: *Define the vocabulary of the blackboard.*

- Elaborate your first definition of the solution space and the abstraction levels of your solutions.
- Find a representation for solutions that allows all knowledge sources to read from and contribute to the blackboard.
- If necessary, provide components that translate between blackboard entries and the internal representations within knowledge sources.

Step6: *Specify the control of the system*

- The 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 *credibility* of a hypothesis is the likelihood that it is correct.

We estimate the credibility of a hypothesis by considering all plausible alternatives to it, and the degree of support each alternative receives from the input data.

Step 7: *Implement the knowledge sources.*

- Split the knowledge sources into condition-parts and action-parts according to the needs of the Control component.
- To maintain the independency and exchangeability of knowledge sources, do not make any assumptions about other knowledge sources or the Control component.
- We can implement different knowledge sources in the same system using different technologies.

Consequences - Advantages

- The Blackboard approach to problem decomposition and knowledge application helps to resolve most of the forces listed in the problem section:
  - Experimentation. In domains in which no closed approach exists and a complete search of the solution space is not feasible, the Blackboard pattern makes experimentation with different algorithms possible

- Support for changeability and maintainability. because the individual knowledge sources, the control algorithm and the central data structure are strictly separated.
- Reusable knowledge sources.

Consequences - Liabilities

- *Difficulty of testing.* Since the computations of a Blackboard system do not follow a deterministic algorithm, its results are often not reproducible.
- *No good solution is guaranteed.* Usually Blackboard systems can solve only a certain percentage of their given tasks correctly.
- *Low Efficiency.* Blackboard systems suffer from computational overheads in rejecting wrong hypotheses.
- *High development effort.* Most Blackboard systems take years to evolve.