USN □□□□□□□□□□

CELEBRATING 25 YEARS
CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Improvement Test – May 2018

Sub:	Software Architecture					**Code:**	10IS81
Date:	21-05-18	Duration:	90 mins	Max Marks:	50	**Sem:** VIII	**Branch:** ISE

Note: Answer **any FIVE** questions, Selecting **at least TWO** questions **from each part**. All questions carry equal marks. Total marks: 50

	PART-A	Marks	OBE CO	OBE RBT
1.	Explain Software Architecture. Explain **ABC**.	[10]	CO1	L4
2.	State the problem of **KWIC**. Propose implicit invocation and Pipes & Filters style to implement a solution for the same.	[10]	CO3	L5
3.	List **Quality Attributes** Scenarios and Explain.	[10]	CO2	L1,L3
4.	Illustrate the behavior of **Blackboard Architecture** based on **Speech recognition** & List the steps to implement Blackboard Pattern.	[10]	CO3	L3,L1

	PART-B	Marks	OBE CO	OBE RBT
5.	Explain the possible dynamic behavior of **MVC pattern**, with suitable sketches	[10]	CO3	L4
6.	List the steps performed in designing an **Architecture using ADD method.** Explain.	[10]	CO4	L1,L3
7.	Explain the **Reflection Architectural Patterns** and its known uses.	[10]	CO4	L5
8.	Discuss the structure, dynamics & Implementation of **Master-Slave pattern**	[10]	CO5	L5

Scheme & Solutions

1. The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. Software architecture is a result of technical, business and social influences. Its existence in turn affects the technical, business and social environments that subsequently influence future architectures. We call this cycle of influences, from environment to the architecture and back to the environment, the Architecture Business Cycle (ABC). This chapter introduces the ABC in detail and examine the following: How organizational goals influence requirements and development strategy.⎫ How requirements lead to architecture.⎫ How architectures are analyzed.⎫ How architectures yield systems that suggest new organizational capabilities and requirements.⎫

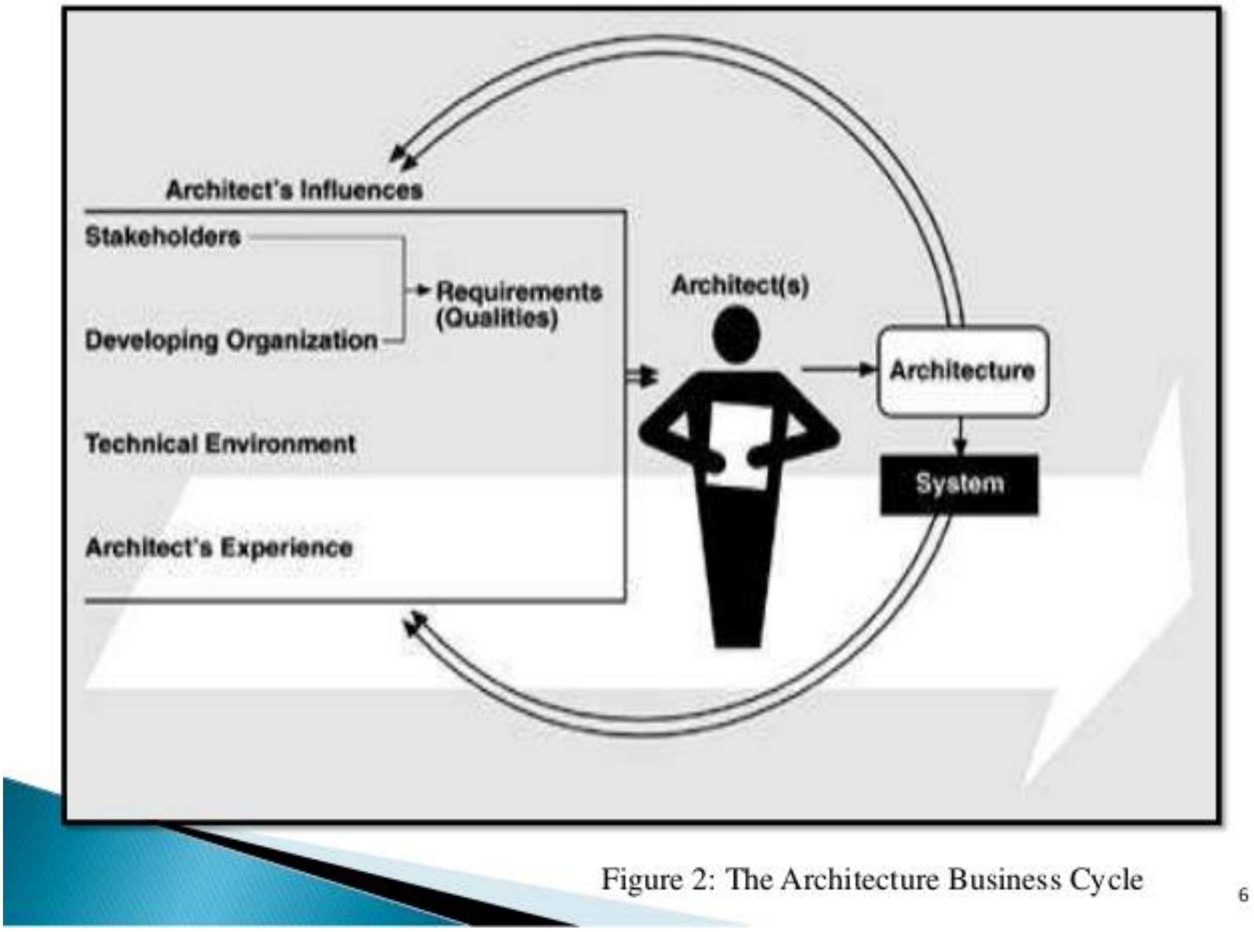


Figure 2: The Architecture Business Cycle

Working of architecture business cycle: 1) The architecture affects the structure of the developing organization. An architecture prescribes a structure for a system it particularly prescribes the units of software that must be implemented and integrated to form the system. Teams are formed for individual software units; and the development, test, and integration activities around the units. Likewise, schedules and budgets allocate resources in chunks corresponding to the units. Teams become embedded in the organization’s structure. This is feedback from the architecture to the developing organization. 2) The architecture can affect the goals of the developing organization. A successful system built from it can enable a company to establish a foothold in a particular market area. The architecture can provide opportunities for the efficient production and deployment of the similar systems, and the organization may adjust its goals to take advantage of its newfound expertise to plumb the market. This is feedback from the system to the developing organization and the systems it builds. 3) The architecture can affect customer requirements for the next system by giving the customer the opportunity to receive a system in a more reliable, timely and economical manner than if the subsequent system were to be built from scratch. 4) The process of system building will affect the architect’s experience with subsequent systems by adding to the corporate experience base. 5) A few systems will influence and actually change the software engineering culture. i.e, The technical environment in which system builders operate and learn.

2. Aims: – To d emonstrate key features of four architectural styles. – To identify relative strengths and weaknesses of these four architectural styles. • First proposed by David Parnas as an example to demonstrate information hiding - key idea b ehind OO. The problem: “The KWIC system index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be “circularly shifted ” by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical ord er.” • Widely used in Computer Science: – Unix man page permutated index – Keyword in context indexes for libraries

KWIC Example Input: Pattern-Oriented Software Architecture Software Architecture Introducing Design Patterns Output (assuming Pattern-Oriented treated as one word): Architecture Software Architecture Pattern-Oriented Software Design Patterns Introducing Introducing Design Patterns Patterns Introducing Design Pattern-Oriented Software Architecture Software Architecture Software Architecture Pattern-Oriented • Can now quickly search for titles that contain phrases such as “Software Architecture” or “Design Pattern” ... 3 Comparison Criteria • Change in overall processing algorithm: line shifting can be performed as line read in, on all lines after they are read, or on demand when sorting requires a new set of shifted lines. • Change in data representation: circular shifts can be stored explicitly or implicitly as indices into the original lines. Different data structures can be used. • Enhancements: eliminate shifts that start with noise words (“a”, “the”), allow deletion of lines, make system interactive • Performance: space and time. • Reuse : to what extent may components be reused? Four well known, published and implemented solutions based on different architectur 4 Main Program/Subroutine with Shared Data 5 Main Program/Subroutine with Shared Data • Four basic functions: Input, Shift, Alphabetise, Output. • Main Program controls these Components and sequences them in turn. • Data is communicated through shared storage: Characters, Index, Alphabetised Index. • Module Input reads the input lines and stores in Characters data structure. • Circular Shift directly accesses Characters to build Index data structure. Each entry in Index identifies the address of circular shift in Characters. • Alphabetiser directly accesses Characters and Index to build Alphabetised Index. This contains an ordered set of indices into Characters. • Output directly accesses Alphabetised Index and Characters to output the ordered, shifted titles. 6 Main Program/Subroutine with Shared Data Strengths: • Efficient use of space (and time) - computations share same data • Intuitive appeal - natural solution? • Enhancements based on shared data easily accommodated e.g. removing shifts starting with noise words. Weaknesses: • Change of data representation affects all modules - all modules take advantage of explicit data representation - no information hiding. • Not particularly supportive of reuse - explicit references to data structures and other functions. Tight coupling. • Changes to overall processing algorithm – depends on nature of change. 7 Abstract Data Type (OO) System I/O Subprogram Call 8 Abstract Data Type (OO) • Similar set of modules to Shared Data Architecture. • Control algorithm is similar. • Key Difference: Data not directly shared - data accessed only through module interfaces. • Circular Shifts and Alphabetic Shifts typically hide shifted/sorted copies of the original lines (although Parnas’ 1972 solution avoided this).

3. Quality attributes are the overall factors that affect run-time behavior, system design, and user experience. They represent areas of concern that have the potential for application wide impact across layers and tiers. Some of these attributes are related to the overall system design, while others are specific to run time, design time, or user centric issues. The extent to which the application possesses a desired combination of quality attributes such as usability, performance, reliability, and security indicates the success of the design and the overall quality of the software application.

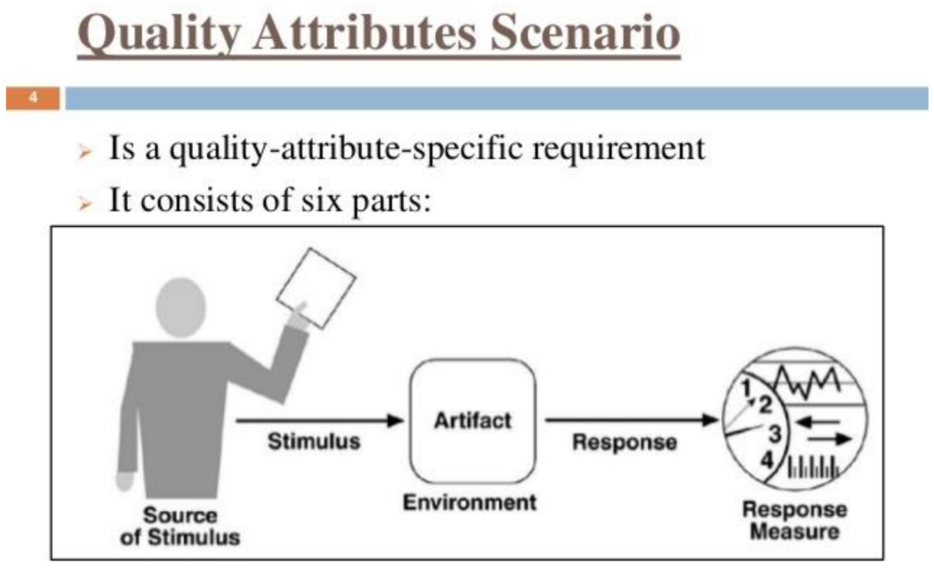
   When designing applications to meet any of the quality attributes requirements, it is necessary to consider the potential impact on other requirements. You must analyze the tradeoffs between multiple quality attributes. The importance or priority of each quality attribute differs from system to system; for example, interoperability will often be less important in a single use packaged retail application than in a line of business (LOB) system.

Common Quality Attributes

The following table describes the quality attributes covered in this chapter. It categorizes the attributes in four specific areas linked to design, runtime, system, and user qualities. Use this table to understand what each of the quality attributes means in terms of your application design.

Category	Quality attribute	Description
Design Qualities	*Conceptual Integrity*	Conceptual integrity defines the consistency and coherence of the overall design. This includes the way that components or modules are designed, as well as factors such as coding style and variable naming.
	Maintainability	Maintainability is the ability of the system to undergo changes with a degree of ease. These changes could impact components, services, features, and interfaces when adding or changing the functionality, fixing errors, and meeting new business requirements.
	Reusability	Reusability defines the capability for components and subsystems to be suitable for use in other applications and in other scenarios. Reusability minimizes the duplication of components and also the implementation time.
Run-time Qualities	*Availability*	Availability defines the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period. Availability will be affected by system errors, infrastructure problems, malicious attacks, and system load.
	Interoperability	Interoperability is the ability of a system or different systems to operate successfully by communicating and exchanging information with other external systems written and run by external parties. An interoperable system makes it easier to exchange and reuse information internally as well as externally.
	Manageability	Manageability defines how easy it is for system administrators to manage the application, usually through sufficient and useful instrumentation exposed for use in monitoring systems and for debugging and performance tuning.
	Performance	Performance is an indication of the responsiveness of a system to execute any action within a given time interval. It can be measured in terms of latency or throughput. Latency is the time taken to respond to any event. Throughput is the number of events that take place within a given amount of time.
	Reliability	Reliability is the ability of a system to remain operational over time. Reliability is measured as the probability that a system will not fail to perform its intended functions over a specified time interval.

	Scalability	Scalability is ability of a system to either handle increases in load without impact on the performance of the system, or the ability to be readily enlarged.
	Security	Security is the capability of a system to prevent malicious or accidental actions outside of the designed usage, and to prevent disclosure or loss of information. A secure system aims to protect assets and prevent unauthorized modification of information.
System Qualities	*Supportability*	Supportability is the ability of the system to provide information helpful for identifying and resolving issues when it fails to work correctly.
	Testability	Testability is a measure of how easy it is to create test criteria for the system and its components, and to execute these tests in order to determine if the criteria are met. Good testability makes it more likely that faults in a system can be isolated in a timely and effective manner.
User Qualities	*Usability*	Usability defines how well the application meets the requirements of the user and consumer by being intuitive, easy to localize and globalize, providing good access for disabled users, and resulting in a good overall user experience.



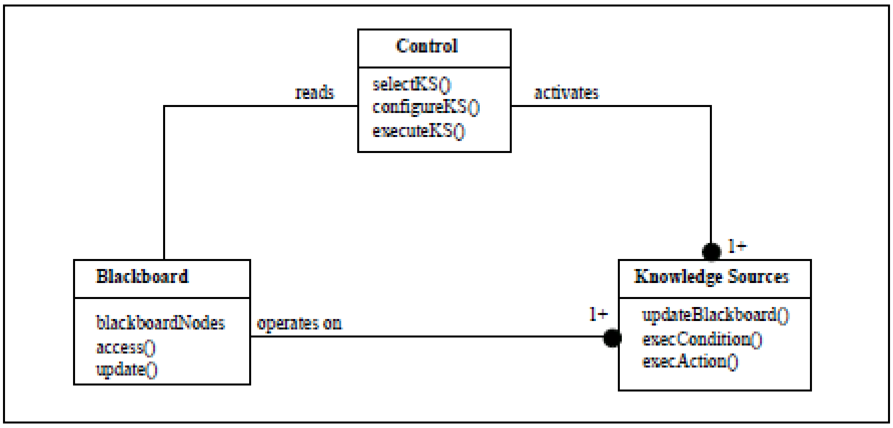
Quality Attributes Scenario

4

- Is a quality-attribute-specific requirement
- It consists of six parts:

Figure 1: Quality attributes Parts

4. Blackboard architecture pattern:



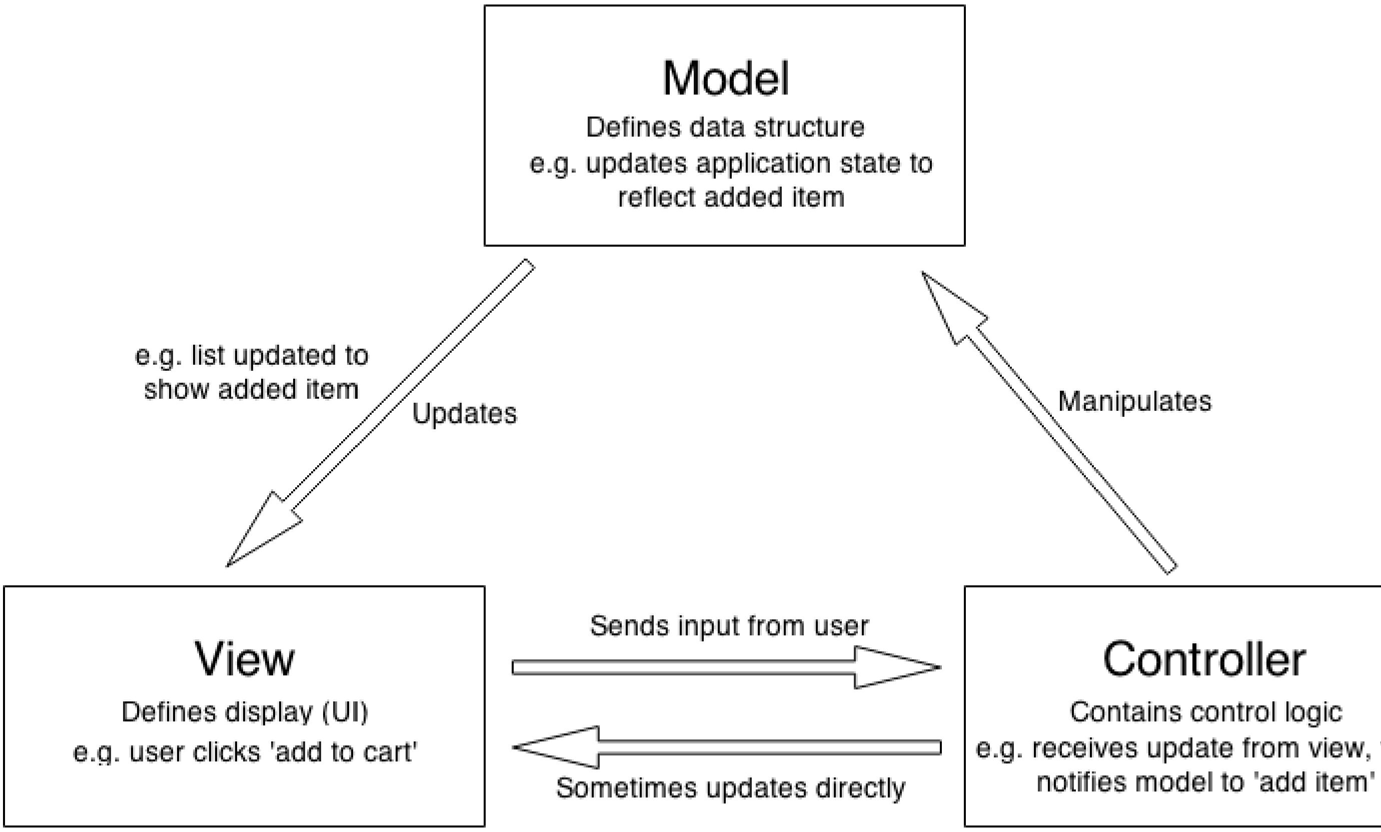
Blackboard: Three major parts: Knowledge sources: Separate, independent parcels of application – dependents knowledge. Blackboard data structure: Problem solving state data, organized into an application-dependent hierarchy Control: Driven entirely by the state of blackboard Invocation of a knowledge source (ks) is triggered by the state of blackboard.⎱ The actual focus of control can be in⎱ - knowledge source - blackboard - Separate module or - combination of these Blackboard systems have traditionally been used for application requiring complex interpretation of signal processing like speech recognition, pattern recognition

The blackboard pattern provides effective solutions for designing and implementing complex systems where heterogeneous modules have to be dynamically combined to solve a problem. This provides non-functional properties such as:

- Reusability
- Changeability
- robustness.[2]

The blackboard pattern allows multiple processes to work closer together on separate threads, polling and reacting when necessary.

5. MVC


Model
Defines data structure
e.g. updates application state to reflect added item
e.g. list updated to show added item
Updates
Manipulates
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sends input from user
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view,
notifies model to 'add item'

MODEL-VIEW-CONTROLLER (MVC) MVC architectural pattern divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change propagation mechanism ensures consistence between the user interface and the model.

The model defines what data the app should contain. If the state of this data changes, then the model will usually notify the view (so the display can change as needed) and sometimes the controller (if different logic is needed to control the updated view).Going back to our shopping list app, the model would specify what data the list items should contain — item, price, etc. — and what list items are already present.

The view defines how the app's data should be displayed.In our shopping list app, the view would define how the list is presented to the user, and receive the data to display from the model.The controller contains logic that updates the model and/or view in response to input from the users of the app.

So for example, our shopping list could have input forms and buttons that allow us to add or delete items. These actions require the model to be updated, so the input is sent to the controller, which then manipulates the model as appropriate, which then sends updated data to the view.You might however also want to just update the view to display the data in a different format, e.g., change the item order to alphabetical, or lowest to highest price. In this case the controller could handle this directly without needing to update the model.

6. a method for designing an architecture to satisfy both quality requirements and functional requirements. We call this method Attribute-Driven Design (ADD). ADD takes as input a set of quality attribute scenarios and employs knowledge about the relation between quality attribute achievement and architecture in order to design the architecture. The ADD method can be viewed as an extension to most other development methods, such as the Rational Unified Process. The Rational Unified Process has several steps that result in the high-level design of an architecture but then proceeds to detailed design

and implementation. Incorporating ADD into it involves modifying the steps dealing with the high-level design of the architecture and then following the process as described by Rational.

ATTRIBUTE-DRIVEN DESIGN

ADD is an approach to defining a software architecture that bases the decomposition process on the quality attributes the software has to fulfill. It is a recursive decomposition process where, at each stage, tactics and architectural patterns are chosen to satisfy a set of quality scenarios and then functionality is allocated to instantiate the module types provided by the pattern. ADD is positioned in the life cycle after requirements analysis and, as we have said, can begin when the architectural drivers are known with some confidence.

The output of ADD is the first several levels of a module decomposition view of an architecture and other views as appropriate. Not all details of the views result from an application of ADD; the system is described as a set of containers for functionality and the interactions among them. This is the first articulation of architecture during the design process and is therefore necessarily coarse grained. Nevertheless, it is critical for achieving the desired qualities, and it provides a framework for achieving the functionality. The difference between an architecture resulting from ADD and one ready for implementation rests in the more detailed design decisions that need to be made. These could be, for example, the decision to use specific object-oriented design patterns or a specific piece of middleware that brings with it many architectural constraints. The architecture designed by ADD may have intentionally deferred this decision to be more flexible.

There are a number of different design processes that could be created using the general scenarios of Chapter 4 and the tactics and patterns of Chapter 5. Each process assumes different things about how to "chunk" the design work and about the essence of the design process. We discuss ADD in some detail to illustrate how we are applying the general scenarios and tactics, and hence how we are "chunking" the work, and what we believe is the essence of the design process.

We demonstrate the ADD method by using it to design a product line architecture for a garage door opener within a home information system. The opener is responsible for raising and lowering the door via a switch, remote control, or the home information system. It is also possible to diagnose problems with the opener from within the home information system.

Sample Input

The input to ADD is a set of requirements. ADD assumes functional requirements (typically expressed as use cases) and constraints as input, as do other design methods. However, in ADD, we differ from those methods in our treatment of quality requirements. ADD mandates that quality requirements be expressed as a set of system-specific quality scenarios. The general scenarios discussed in Chapter 4 act as input to the requirements process and provide a checklist to be used in developing the system-specific scenarios. System-specific scenarios should be defined to the detail necessary for the application. In our examples, we omit several portions of a fully fleshed scenario since these portions do not contribute to the design process.

For our garage door example, the quality scenarios include the following:

- The device and controls for opening and closing the door are different for the various products in the product line, as already mentioned. They may include controls from within a home information system. The product architecture for a specific set of controls should be directly derivable from the product line architecture.
- The processor used in different products will differ. The product architecture for each specific processor should be directly derivable from the product line architecture.
- If an obstacle (person or object) is detected by the garage door during descent, it must halt (alternately re-open) within 0.1 second.
- The garage door opener should be accessible for diagnosis and administration from within the home information system using a product-specific diagnosis protocol. It should be possible to directly produce an architecture that reflects this protocol.

Beginning ADD

We have already introduced architectural drivers. ADD depends on the identification of the drivers and can start as soon as all of them are known. Of course, during the design the determination of which architectural drivers are key may change either as a result of better understanding of the requirements or as a result of changing requirements. Still, the process can begin when the driver requirements are known with some assurance.

In the following section we discuss ADD itself.

ADD Steps

We begin by briefly presenting the steps performed when designing an architecture using the ADD method. We will then discuss the steps in more detail.

1. Choose the module to decompose. The module to start with is usually the whole system. All required inputs for this module should be available (constraints, functional requirements, quality requirements).
2. Refine the module according to these steps:
    a. Choose the architectural drivers from the set of concrete quality scenarios and functional requirements. This step determines what is important for this decomposition.
    b. Choose an architectural pattern that satisfies the architectural drivers. Create (or select) the pattern based on the tactics that can be used to achieve the drivers. Identify child modules required to implement the tactics.
    c. Instantiate modules and allocate functionality from the use cases and represent using multiple views.
    d. Define interfaces of the child modules. The decomposition provides modules and constraints on the types of module interactions. Document this information in the interface document for each module.
    e. Verify and refine use cases and quality scenarios and make them constraints for the child modules. This step verifies that nothing important was forgotten and prepares the child modules for further decomposition or implementation.
3. Repeat the steps above for every module that needs further decomposition.

1 Choose the Module to Decompose

The following are all modules: system, subsystem, and submodule. The decomposition typically starts with the system, which is then decomposed into subsystems, which are further decomposed into submodules.

In our example, the garage door opener is the system. One constraint at this level is that the opener must interoperate with the home information system.

2.a Choose the Architectural Drivers

As we said, architectural drivers are the combination of functional and quality requirements that "shape" the architecture or the particular module under consideration. The drivers will be found among the top-priority requirements for the module.

In our example, the four scenarios we have shown are architectural drivers. In the systems on which this example is based, there were dozens of quality scenarios. In examining them, we see a requirement for real-time performance,[1] and modifiability to support product lines. We also see a requirement that online diagnosis be supported. All of these requirements must be addressed in the initial decomposition of the system.

[1] A 0.1-second response when an obstacle is detected may not seem like a tight deadline, but we are discussing a mass market where using a processor with limited power translates into substantial cost savings. Also, a garage door has a great deal of inertia and is difficult to stop.

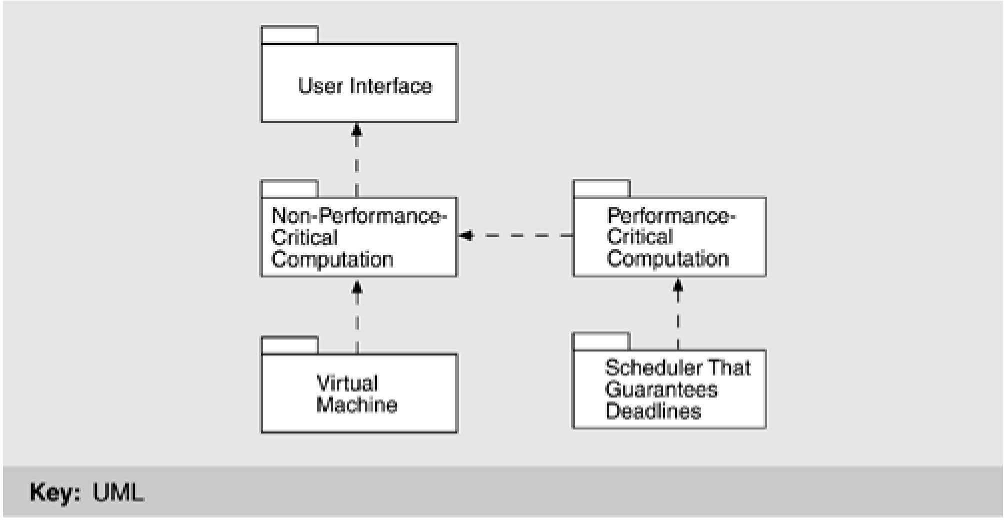
The determination of architectural drivers is not always a top-down process. Sometimes detailed investigation is required to understand the ramifications of particular requirements. For example, to determine if performance is an issue for a particular system configuration, a prototypical implementation of a piece of the system may be required. In our example, determining that the performance requirement is an architectural driver requires examining the mechanics of a garage door and the speed of the potential processors.

We will base our decomposition of a module on the architectural drivers. Other requirements apply to that module, but, by choosing the drivers, we are reducing the problem to satisfying the most important ones. We do not treat all of the requirements as equal; the less important requirements are satisfied within the constraints of the most important. This is a significant difference between ADD and other architecture design methods.

2.b Choose an Architectural Pattern

As discussed in Chapter 5, for each quality there are identifiable tactics (and patterns that implement these tactics) that can be used in an architecture design to achieve a specific quality. Each tactic is designed to realize one or more quality attributes, but the patterns in which they are embedded have an impact on other quality attributes. In an architecture design, a composition of many such tactics is used to achieve a balance between the required multiple qualities. Achievement of the quality and functional requirements is analyzed during the refinement step.

The goal of step 2b is to establish an overall architectural pattern consisting of module types. The pattern satisfies the architectural drivers and is constructed by composing selected tactics. Two main factors guide tactic selection. The first is the drivers themselves. The second is the side effects that a pattern implementing a tactic has on other qualities.


User Interface
Non-Performance-Critical Computation
Performance-Critical Computation
Virtual Machine
Scheduler That Guarantees Deadlines
Key: UML

7. The Reflection architectural pattern provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects¬ such as type structures and function call mechanisms. In this pattern, an application is split into two parts. A¬ meta level provides information about selected system properties and makes the software self-aware. A base level includes the application logic. Its¬ implementation builds on the meta level. Changes to information kept in the meta level affect¬ subsequent base-level behavior.

   Support for variation is the key to sustainable architectures for long-lived applications. – Over time they must respond to evolving and changing technologies, requirements, and platforms. However, it is hard to forecast what can vary in an application and¬ when it must respond to a specific variation request. The need for variation can occur at any time, specifically while the¬ application is in productive use. Variations can also be of any scale, ranging from local adjustments¬ of an algorithm to fundamental modifications of distribution infrastructure. The complexity associated with particular variations should be¬ hidden from maintainers, and there should be a uniform mechanism for supporting different types of variation.

   Designing a system that meets a wide range of¬ different requirements a priori can be an overwhelming task. A better solution is to specify an architecture¬ that is open to modification and extension. The resulting system can then be adapted to¬ changing requirements on demand. In other words, we want to design for change¬ and evolution.

   Changing software is tedious, error prone, and often expensive. – Wide-ranging modifications usually spread over many components and even local changes within one component can affect other parts of the system. – Every change must be implemented and tested carefully. – Software which actively supports and controls its own modification can be changed more effectively and more safely. Adaptable software systems usually have a complex¬ inner structure. – Aspects that are subject to change are encapsulated within separate components. – The implementation of application services is spread over many small components with different interrelationships. – To keep such systems maintainable, we prefer to hide this complexity from maintainers of the system.

The more techniques that are necessary for keeping a system changeable, such as parameterization, subclassing, or even copy and paste, the more awkward and complex its modification becomes. – A uniform mechanism that applies to all kinds of changes is easier to use and understand. Changes can be of any scale, from providing shortcuts¬ for commonly-used commands to adapting an application framework for a specific customer. Even fundamental aspects of software systems can¬ change, for example the communication mechanisms between components.

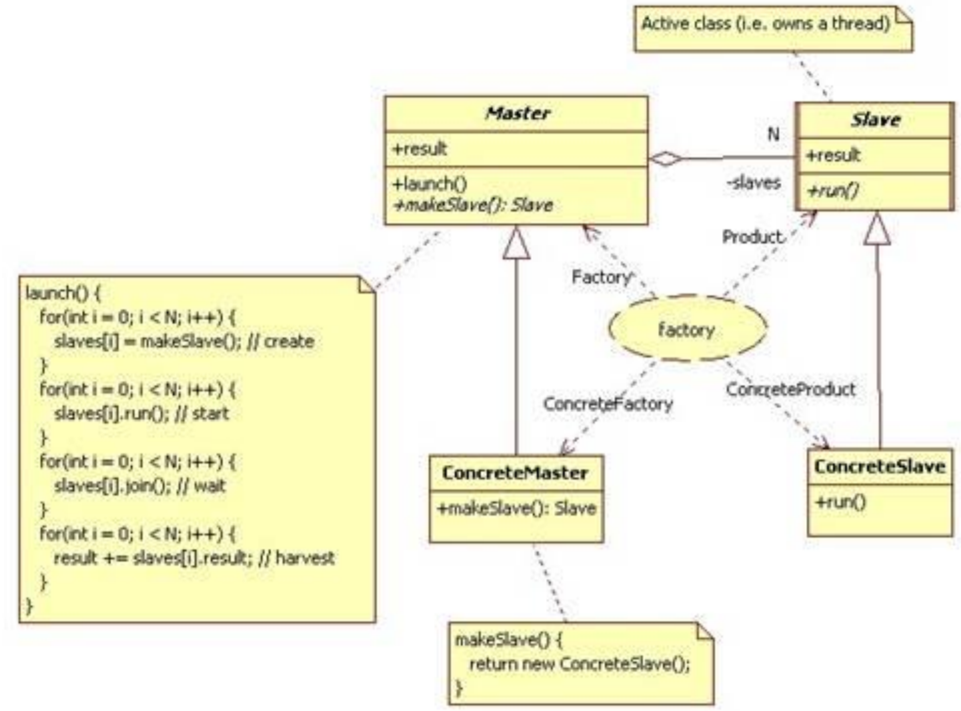
Solution Encapsulate information about properties and¬ variant aspects of the application's structure, behavior, and state into a set of meta-objects. Separate the meta-objects from the core¬ application logic via a two-layer architecture: – The meta level contains the meta-objects – The base level contains the application logic. Base-level objects consult an appropriate metaobject¬ before they execute behavior or access state that potentially can vary.

Solution (2) The meta level provides a self-representation of the¬ software to give it knowledge of its own structure and behavior, and consists of so-called meta-objects. Meta-objects encapsulate and represent information¬ about the software. Examples include type structures, algorithms, or even function call mechanisms. The base level defines the application logic. Its¬ implementation uses the meta-objects to remain independent of those aspects that are likely to change. – For example, in a distributed application, base-level components might only communicate with each other via a meta-object that implements a specific user-defined messaging mechanism. – Changing this meta-object changes the way in which base-level components communicate, but without modifying the baselevel code.

Meta-Object Protocol The meta level also implements a meta-object¬ protocol (MOP), which is a specialized interface that administrators, maintainers, or even other systems can use to dynamically configure and modify the meta-objects in well defined way. Since the base-level implementation explicitly¬ builds upon information and services provided by meta-objects, changing them has an immediate effect on the subsequent behavior of the base level.

Meta-Object Protocol (2) When extending the software, you pass the new code to¬ the meta level as a parameter of the meta-object protocol. The meta-object protocol itself is responsible for¬ integrating all change requests. – It performs modifications and extensions to meta-level code, and if necessary re-compiles the changed parts and links them to the application while it is executing. This provides a reflective application with explicit¬ control over its own modification.

8. MS pattern



```
launch() {
  for(int i = 0; i < N; i++) {
    slaves[i] = makeSlave(); // create
  }
  for(int i = 0; i < N; i++) {
    slaves[i].run(); // start
  }
  for(int i = 0; i < N; i++) {
    slaves[i].join(); // wait
  }
  for(int i = 0; i < N; i++) {
    result += slaves[i].result; // harvest
  }
}
```

```
makeSlave() {
  return new ConcreteSlave();
}
```

The Master-Slave pattern is often used for multi-threaded applications in which many instances of the same problem must be solved. (Travelling Salesman Problem, for example.) The master creates and launches slaves to solve these instances in "parallel". When all of the slaves have finished, the master harvests the results.

Master-Slave pattern is also used for user interfaces and servers. In both cases the master listens for commands coming either from the user or from clients. When a command is received, a slave is launched to execute the command while the master resumes listening for more commands (such as the "suspend the last command" command.)