Module 3

Problem statement

The first step to develop anything is to state the requirements. Being vague about your objective only postpones decisions to a later stage where changes are much costly. The problem statement should state what is to be done and not how it is to be done. It should be a statement of needs, not a proposal for a solution. The requestor should indicate which features are mandatory and which are optional; to avoid overly constraining design decisions. The requestor should avoid describing system internals.

A problem statement may have more or less detail. Most problem statements are ambiguous, incomplete or even inconsistent. Some requirements are wrong. Some requirements have unpleasant consequences on the system behavior or impose unreasonable implementation costs. Some requirements seem reasonable first. The problem statement is just a starting point for understanding problem. Subsequent analysis helps in fully understanding the problem and its implication.

Object Modeling

The first step in analyzing the requirements is to construct an object model. It describes real world object classes and their relationships to each other. Information for the object model comes from the problem statement, expert knowledge of the application domain, and general knowledge of the real world. If the designer is not a domain expert, the information must be obtained from the application expert and checked against the model repeatedly. The object model diagrams promote communication between computer professionals and application domain experts.

The steps for object modeling are:

1. ***Identifying Object Classes***

The first step in constructing an object model is to identify relevant object classes from the application domain. The objects include physical entities as well as concepts. All classes must make sense in the application domain; avoid computer

implementation constructs such as linked lists. Not all classes are explicit in the problem statement; some are implicit in the application domain or general knowledge.

2. ***Keeping the Right classes***

Now discard the unnecessary and incorrect classes according to the following criteria:

- Redundant Classes - If two classes express the same information, the most descriptive name should be kept.
- Irrelevant Classes - If a class has little or nothing to do with the problem, it should be eliminated .this involves judgment, because in another context the class could be important.
- Vague Classes - A class should be specific .Some tentative classes may have ill-defined boundaries or be too broad in scope.
- Attributes - Names that primarily describe individual objects should be restated as attributes. If independent existence of a property is important, then make it a class and not an attribute.
- Operations - If a name describes an operation that is applied to objects and not manipulated in its own right, then it is not a class.
- Roles - The name of a class should reflect its intrinsic nature and not a role that it plays in an association.
- Implementation Constructs - Constructs extraneous to real world should be eliminated from analysis model. They may be needed later during design, but not now.

3. ***Preparing a data dictionary***

Prepare a data dictionary for all modeling entities .Write a paragraph precisely describing each object class. Describe the scope of the class within the current problem, including any assumptions or restrictions on its membership or use. Data dictionary also describes associations, attributes and operations.

4. ***Identifying Associations***

Identify associations between classes .Any dependency between two or more classes is an association. A reference from one class to another is an association. Associations often correspond to stative verbs or verb phrases. These include physical location (next to, part of), directed actions (drives), communication (talks to), ownership (has, part of) or satisfaction of some kind (works for). Extract all the candidates from the problem statement and get them down on to the paper first, don't try to refine things too early. Don't spend much time trying to distinguish between association and aggregation. Use whichever seems most natural at the time and moves on.

Majority of the associations are taken from verb phrases in the problem statement. For some associations, verb phrases are implicit. Some associations depend on real world knowledge or assumptions. These must be verified with the requestor, as they are not in the problem statement.

5. Keeping the right Associations

Discard unnecessary and incorrect associations using the following criteria:

- Associations between eliminated classes

If one of the classes in the association has been eliminated, then the association must be eliminated or restated in terms of other classes.

- Irrelevant or implementation Associations

Eliminate any association that are outside the problem domain or deal with implementation constructs.

- Actions

An association should describe a structural property of the application domain, not a transient event.

- Ternary Association

Most associations between three or more classes can be decomposed into binary associations or phrased as qualified associations. If a term ternary association is purely descriptive and has no features of its own, then the term is a link attribute on a binary association.

- Derived Association

Omit associations that can be defined in terms of other associations because they are redundant. Also omit associations defined by conditions on object attributes.

- Misnamed Associations

Don't say how or why a situation came about, say what it is. Names are important to understanding and should be chosen with great care.

- Role names

Add role names where appropriate. The role name describes the role that a class in the association plays from the point of view of other class.

- Qualified Associations

Usually a name identifies an object within some context; mostly names are not globally unique. The context combines with the name to uniquely identify the object.

A qualifier distinguishes objects on the "many" side of an association.

- Multiplicity

Specify multiplicity, but don't put too much effort into getting it right, as multiplicity often changes during analysis.

- Missing Associations

Add any missing associations that are discovered.

6. Identifying Attributes

Identify object attributes. Attributes are properties of individual objects, such as name, weight, velocity, or color. Attributes shouldn't be objects, use an association to show any relationship between two objects. Attributes usually correspond to nouns followed by possessive phrases, such as the 'color of the car'. Adjectives often represent specific enumerated attribute values, such as red. Unlike classes and associations, attributes are less likely to be fully described in the problem statement. You must draw on your knowledge of the application domain and the real world to find them. Attributes seldom affect the basic structure of the problem. Only consider attributes that directly relate to a particular application. Get the most important attributes first, fine details can be added

later. Avoid attributes which are solely for implementation. Give each attribute a meaningful name. Derived attributes are clearly labeled or should be omitted. e.g.:- age. Link attributes should also be identified. They are sometimes mistaken for object attributes.

7. Keeping the Right Attributes

Eliminate unnecessary and incorrect attributes with the following criteria:

- Objects: If the independent existence of an entity is important, rather than just its value, then it is an object.

  e.g.:- Boss is an object and salary is an attribute.

The distinction often depends on the application .An entity that has features of its own within the given application is an object.

- Qualifiers: If the value of an attribute depends on a particular context, then consider restating the attribute as a qualifier. e.g.:- Employee number is not a unique property of a person with two jobs, it qualifies the association company employs person.
- Nouns: Names are often better modeled as qualifiers rather than attributes.
- Identifiers: Object Oriented Languages incorporate the notion of an object id for unambiguously referencing an object. Do not list these object identifiers in object models, as object identifiers are implicit in object models. One list attributes which exist in application domain.
- Link Attributes: If a property depends on the presence of a link, then the property is an attribute of the link and not of related objects. Link attributes are usually obvious on many to many associations, they can't be attached to the "many" objects without losing information. Link attributes are also subtle on one-one association.
- Internal values: If an attribute describes the internal state that is invisible outside the object, then eliminate it from the analysis.
- Fine detail: Omit minor attributes which are unlikely to affect most operations.
- Discordant attributes: An attribute that seems completely different from and unrelated to all other attributes may indicate a class that should be split into two distinct

classes.

8. Refining with Inheritance

The next step to organize classes by using inheritance to share common structure .inheritance can be added in two directions:- by generalizing common aspects of existing classes into a superclass (bottom up),refining existing classes into specialized subclasses (top down).

9. Testing Access Paths

Trace access paths through object model diagram to see if they yield sensible results. Where a unique value is expected is there a path yielding a unique result? For multiplicity is there a way to pick out unique values when needed? Are there useful questions which can't be answered? Thus we will be able to identify if there is some missing information.

10. Iterating Object Model

Object model is rarely correct after a single pass. Continuous iteration is needed. Different parts of a model are iterated often at different stages of completion. If there is any deficiency, go aback to an earlier stage to correct it. Some refinements are needed only after the functional and dynamic models.

11. Grouping Classes into Modules

Group the classes into sheets and modules. Reuse module from previous design.

Dynamic Modeling:

Dynamic Modeling shows the time-dependent behavior of the system and the objects in it. Begin dynamic analysis by looking for events-externally visible stimuli and responses. Summarize permissible event sequences for each object with a state diagram. Dynamic

model is insignificant for a purely static data repository. It is important in the case of interactive systems. For most problems, logical correctness depends on the sequences of interactions, not the exact time of interactions. Real time systems do have specific timing requirements on interactions that must be considered during analysis.

The following steps are performed in constructing a dynamic model:

(i) Preparing a Scenario

Prepare one or more typical dialogs between user and system to get a feel for expected system behavior. These scenarios show the major interactions, external display format and information exchanges.

Approach the dynamic model by scenarios, to ensure that important steps are not overlooked and that the overall flow of the interaction is smooth and correct. Sometimes the problem statement describes the full interaction sequence, but most of the time you will have to invent the interaction format.

- The problem statement may specify needed information but leaves open the manner in which it is obtained.
- In many applications gathering information is a major task or sometimes the only major task. The dynamic model is critical in such applications.
- First prepare scenarios for "normal " cases , interactions without any unusual inputs or error conditions
- Then consider "special "cases, such as omitted input sequences, maximum and minimum values, and repeated values.
- Then consider user error cases, including invalid values and failures to respond. For many interactive applications, error handling is the most different part of the implementation. If possible, allow the user to abort an operation or roll back to a well defined starting point at each step.
- Finally consider various other kinds of interactions that can be overlaid on basic interactions, such as help requests and status queries.
- A scenario is a sequence of events .An event occurs when information is

exchanged between an object in the system and an outside agent, such as user. The values exchanged are parameters of the event. The events with no parameters are meaningful and even common. The information in such event is the fact that it has occurred a pure signal. Anytime information is input to the system or output from the system.

- For each event identify the actor (system, user or other external agent) that caused the event and the parameters of the event.
- The screen layout or output format generally doesn't affect the logic of the interaction or the values exchanged.
- Don't worry about output formats for the initial dynamic model; describe output formats during refinement of the model.

(ii) Interface Formats

Most interactions can be separated into two parts:

- application logic
- user interface

Analysis should concentrate first on the information flow and control, rather than the presentation format. Same program logic can accept input from command lines, files, mouse buttons, touch panels, physical push buttons, or carefully isolated. Dynamic model captures the control logic of the application. It is hard to evaluate a user interface without actually testing it .Often the interface can be mocked up so that users can try it. Application logic can often be simulated with dummy procedures. Decoupling application logic from user interface allows "look and feel" of the user interface to be evaluated while the application is under development.

i. **Identifying Events**

Examine the scenario to identify all external events. Events include all signals, inputs, decisions, interrupts, transitions, and actions to or from users or external devices. Internal computation steps are not events, except for decision points that interact with the external world .Use scenarios to find normal events, but don't forget error conditions and unusual events. An action by an object that transmits information is an event. Most object to object interaction and operations correspond to events. Some information flows are

implicit. Group together under a single name event that have the same effect on flow of control, even if their parameter values differ. You must decide when differences in quantitative values are important enough to distinguish. The distinction depends on the application. You may have to construct the state diagram before you can classify all events; some distinctions between events may have no effect on behavior and can be ignored. Allocate each type of event to the object classes that send it and receive it. The event is an output event for the sender and an input event for the receiver. Sometimes an object sends an event to itself, in which case the event is both an output and input for the same class. Show each scenario as an event trace – an ordered list of events between different objects assigned to columns in a table. If more than one object of the same class participates in the scenario, assign a separate column to each object. By scanning a particular column in the trace, you can see the events that directly affect a particular object. Only these events can appear in the state diagram for the object. Show the events between a group of classes (such as a module) on an event flow diagram. This diagram summarizes events between classes, without regard for sequence. Include events from all scenarios, including error events. The event flow diagram is a dynamic counterpart to an object diagram. Paths in the object diagram show possible information flows; paths in the event flow diagram show possible control flows.

(iv) Building a State Diagram

Prepare a state diagram for each object class with non trivial dynamic behavior , showing the events the object receives and sends .Every scenario or event trace corresponds to path through the state diagram .Each branch in a control flow is represented by a state with more than one exit transition. The hardest thing is deciding at which state an alternative path rejoins the existing diagram. Two paths join at a state if the object "forgets" which the one was taken. In many cases, it is obvious from your knowledge of the application that two states are identical. Beware of two paths that appear identical but which can be distinguished under some circumstances. The judicious use of parameters and conditional transitions can simplify state diagrams considerably but at the cost of mixing together state information and data. State diagrams with too much data

dependency can be confusing and counterintuitive. Another alternative is to partition a state diagram into two concurrent subdiagrams, using one subdiagram for the main line and the other for the distinguishing information. After normal events have been considered, add boundary cases and special cases. Handling user errors needs more thought and code than normal case. You are finished with the state diagram of a class when the diagram handles all events that can affect an object of the class in each of its states. If there are complex interactions with independent inputs, we use nested state diagrams. Repeat the above process of building state diagrams for each class of objects. Concentrate on classes with important interactions. Not all classes need state diagrams. Many objects respond to input events independent of their past history, or capture all important history as parameters that do not affect control. Such objects may receive and send events. List the input events for each object and output event sent in response to each input event, but there will be no further state structure. Eventually you may be able to write down state diagram directly without preparing event traces.

(v) Matching Events between Objects.

Check for completeness and consistency at the system level when the state diagram for each class is complete .Every event should have a sender and a receiver, occasionally the same object .States without predecessors or successors are suspicious; make sure they represent starting or termination points of the interaction sequences .Follow the effects of an input event from object to object through the system to make sure that they match the scenario. Objects are inherently concurrent; beware of synchronization errors where an input occurs at an awkward time. Make sure that corresponding events on different state diagrams are consistent. The set of state diagrams for object classes with important dynamic behavior constitute the dynamic model for the application.

Functional Modeling

The functional model shows how values are computed, without regard for sequencing, decisions or object structure. It shows which values depend on which other values and which function relate them. The dfds are useful for showing functional dependencies

.Functions are expressed in various ways, including natural language, mathematical equations and pseudo code. Processes on dfd corresponds to activities or actions in the state diagrams of the class. Flows on dfd correspond to objects or attribute values on attribute values in object diagram. Construct the functional model after the two models. The following steps are performed in constructing functional model:

(i) Identifying input and output values

List input and output values. The input and output values are parameters of events between system and outside world. Find any input or output values missed.

(ii) Building DFD

Now construct a DFD, show each output value is completed from input values. A DFD is usually constructed in layers. The top layer may consist of a single process, or perhaps one process to gather inputs, compute values, and generate outputs. Within each dfd layer, work backward from each output value to determine the function that computes it. If the inputs are operations are all inputs of the entire diagram, you are done; otherwise some of the operation inputs are intermediate values that must be traced backward in turn. You can also trace forward from inputs to outputs, but it is usually harder to identify all users of an input, than to identify all the sources of an output.

- Expand each non trivial process in the top level DFD into a lower level DFD.
- Most systems contain internal storage objects that retain values between iterations. An internal store can be distinguished from a data flow or a process because it receives values that don't result in immediate outputs but are used at some distant time in the future.
- The DFD show only dependencies among operations. They don't show decisions.(control flows)

iii) Describing Functions

When the dfd has been refined enough, write a description of each function. The description can be in natural language, mathematical equations, pseudocode, decision tables, or some appropriate form.

- Focus on what the function does, and not how it is implemented.

- The description can be declarative or procedural.
- A declarative description specifies the relationships between input and output values and the relationships among output values.

For example, “sort and remove duplicate values”

Declarative: “Every value in the input list appears exactly once in the output list, and the values in the output list are in strictly increasing order.”

Procedural:-specifies function by giving an algorithm to compute it. The purpose of the algorithm is only to specify what the function does.

A declarative description is preferable; they don’t imply implementation but if procedural is easy it should be used.

iv) Identify constraints between objects

Identify constraints between objects. Constraints are functional dependencies between objects that are not related by an input output dependency. Constraints can be on two objects at the same time, between instances of the same object at different times (an invariant) or between instances of different object at different times.

-> Preconditions on functions are constraints that the input values must satisfy.

-> Post conditions are constraints that the output values are guaranteed to hold.

-> State the times or conditions under which the constraints hold.

v) Specifying Optimization criteria

Specify values to be maximized, minimized or optimized. If there is several optimization criteria that conflict, indicate how the trade-off is to be decided.

System Design

System design is the high-level strategy for solving the problem and building a solution.

The steps involved in system design are:

1. ESTIMATING SYSTEM performance

Prepare a rough performance estimate. The purpose is not to achieve high accuracy, but

merely to determine if the system is feasible.

2. Making a reuse plan

There are 2 different aspects of reuse – using existing things and creating reusable new things. It is much easier to reuse existing things than to design new things. Reusable things include models, libraries, patterns, frameworks etc.

3. BREAKING A SYSTEM INTO SUBSYSTEMS

First step of system design is to divide the system in a small number of subsystems. Each subsystem encompasses aspects of the system that share some common property such as similar functionality, same physical location, execution on same kind of hardware.

- A subsystem is a package of classes, associations, operations, events and constraints interrelated and have a well defined small interface with other subsystems. Subsystems are identified by the services provided. Determining subsystems allow a division of labor so each can be designed by a separate team.
- Most interactions should be within subsystems rather than across boundaries and it reduces dependencies among subsystems.
- Relationship between two subsystems can be client-supplier or peer-to-peer. In a client-supplier relationship, the client calls on the supplier for service. Client must know the interface of the supplier but not vice versa. In a peer-to-peer relationship, each can call on the other and therefore they must know each others interfaces. One way communication (client-supplier) is much easier to build, understand and change than two way (Peer-to-Peer).
- The decomposition of systems into subsystems may be organized as a sequence of layers or partitions.
- Layered system is an ordered set of virtual worlds, a layer knows about the layer below but not above. A client supplier relationship exists between lower layers and upper layers.
- Partitions divide into several independent SS, each providing a type of service. A

system can be a hybrid of layered and partitioned

4. CHOOSING THE SYSTEM TOPOLOGY

System Topology defines the flow of information among subsystems. The flow may be a pipeline (compiler) or star which has a master that controls all other interactions. Try to use simple topologies when possible to reduce the number of interactions between subsystems.

- In analysis as in the real world and in hardware all objects are concurrent. In implementation not all software objects are concurrent because one processor may support many objects. An important goal of system design is to identify which object must be active concurrently and which objects have activity that is mutually exclusive (not active at the same time). The latter objects can be folded into a single thread of control or task (runs on a single processor).

5. IDENTIFYING CONCURRENCY

- Dynamic Modeling is the guide to concurrency - Two objects are said to be inherently concurrent if they can receive events at the same time without interacting. If the events are unsynchronized, the objects cannot be folded into a single thread of control. (engine and wing control on an airplane must operate concurrently)
- Two subsystems that are inherently concurrent need not necessarily be implemented as separate hardware units. The purpose of interrupts, Operating System and tasking mechanism is to simulate logical concurrency in a uniprocessor - so if there are no timing constraints on the response then a multitasking Operating System can handle the computation.
- All objects are conceptually concurrent, but many objects are interdependent. By examining the state diagram of individual objects and the exchange of events between them, many objects can be folded into a single thread of control. A thread of control is a path through a set of state diagrams on which only a single object at a time is active - thread remains in a state diagram until an object sends an event to another object and waits for another event.
- On each thread of control only a single object is active at a time. Threads of

control are implemented as tasks in computer systems.

6. ESTIMATING HARDWARE RESOURCE REQUIREMENTS

- The decision to use multiple processors or hardware functional units is based on the need for higher performance than a single CPU can provide (or fault tolerant requests). Numbers of processors determine the volume of computations and speed of the machine.
- The system designer must estimate the required CPU power by computing the steady state load as the product of the number of transactions per second and the time required to process a transaction. Experimentation is often useful to assist in estimating.

HARDWARE/SOFTWARE TRADEOFFS

Hardware can be viewed as a rigid but highly optimized form of software. The system designer must decide which subsystems will be implemented in hardware and which in software.

Subsystems are in hardware for 2 reasons:

- Existing hardware provides exactly the functionality required – e.g. floating point processor
- Higher performance is required than a general purpose CPU can provide.

Compatibility, cost, performance, flexibility are considerations whether to use hardware or software. Software in many cases may be more costly than hardware because the cost of people is very high. Hardware of course is much more difficult to change than software.

7. ALLOCATING TASKS TO PROCESSORS

Tasks are assigned to processors because:

- Certain tasks are required at specific physical locations
- Response time or information flow rate exceeds the available communication bandwidth between a task and a piece of hardware.

- Computation rates are too great for single processor, so tasks must be spread among several processors.

8. MANAGEMENT OF DATA STORES

- The internal and external data stores in a system provide clean separation points between subsystems with well defined interfaces. Data stores may combine data structures, files, and database implemented in memory or on secondary storage devices. The different data stores provide trade-offs between cost, access time, capacity and reliability.
- Files are cheap, simple and permanent form of data store. File operations are low level and applications must include additional code to provide a suitable level of abstraction. File implementations vary from machine to machine so portability is more difficult.
- Databases are powerful and make applications easier to port. They do have a complex interface and may work awkwardly with programming languages.

The kind of data that belongs in a Database:

- Data that requires access at fine levels of detail by multiple users
- Data that can be efficiently managed by DBMS commands
- Data that must be ported across many hardware and operating system platforms
- Data that must be accessible by more than one application program.

The kind of data that belongs in a file:

- Data that is voluminous but difficult to structure within the confines of a Database (graphics bit map)
- Data that is voluminous and of low information density (debugging data, historical files)
- Raw data that is summarized in the Database
- Volatile data

ADVANTAGES OF USING A Database

- Handles crash recovery, concurrency, integrity
- Common interface for all applications
- Provides a standard access language - SQL

DISADVANTAGES OF USING A Database

- Performance overhead
- Insufficient functionality for advanced applications - most databases are structured for business applications.
- Awkward interface with programming languages.

9. HANDLING GLOBAL RESOURCES

- The system designer must identify the global resources and determine mechanisms for controlling access to them - processors, tape drives, etc
- If the resource is a physical object, it can control itself by establishing a protocol for obtaining access within a concurrent system. If the resource is logical entity (object ID for example) there is a danger of conflicting access. Independent tasks could simultaneously use the same object ID. Each global resource must be owned by a guardian object that controls access to it.

10. CHOOSING Software CONTROL IMPLEMENTATION

There are two kinds of control flows in a software system: external control and internal control. Internal control is the flow of control within a process. It exists only in the implementation and therefore is not inherently concurrent or sequential. External control is the flow of externally visible events among the objects in the system.

There are three kinds of external control:

- Procedure driven systems: control resides within program code; it is easy to implement but requires that concurrency inherent in objects be mapped into a sequential flow of control.

- Event driven systems - control resides within a dispatcher or monitor provided by the language, subsystem or operating system. Application procedures are attached to events and are called by the dispatcher when the corresponding events occur.
- Concurrent systems: control resides concurrently in several independent objects, each a separate task.

11. HANDLING BOUNDARY CONDITIONS

We must be able to handle boundary conditions such as initialization, termination and failure.

1. Initialization: The system must be brought from a quiescent initial state to a sustainable steady state condition. Things to be initialized include constant data, parameters, global variables, tasks, guardian objects and possibly the class hierarchy itself.

2. Termination: It is usually simpler than initialization because many internal objects can simply be abandoned. The task must release any external resources that it had reserved. In a concurrent system, one task must notify other tasks of its termination.

3. Failure: It is the unplanned termination of a system. Failure can arise from user errors, from the exhaustion of system resources, or from an external breakdown.

12. SETTING TRADE-OFF PRIORITIES

Cost, maintainability, portability, understandability, speed, may be traded off during various parts of system design.

13. COMMON ARCHITECTURAL FRAMEWORKS

There are several prototypical architectural frameworks that are common in existing systems. Each of these is well-suited to a certain kind of system. If we have an application with similar characteristics, we can save effort by using the corresponding architecture, or at least use it as a starting point for our design. The kinds of systems

include:

1. Batch transformation: It is a data transformation executed once on an entire input set. A batch transformation is a sequential input-to-output transformation, in which inputs are supplied at the start, and the goal is to compute an answer. There is no ongoing interaction with the outside world. The most important aspect of the batch transformation is the functional model which specifies how input values are transformed into output values. The steps in designing a batch transformation are:

1. Break the overall transformation into stages, each stage performing one part of the transformation. The system diagram is a data flow diagram. This can usually be taken from the functional model.
2. Define intermediate object classes for the data flows between each pair of successive stages.
3. Expand each stage in turn until the operations are straightforward to implement.
4. Restructure the final pipeline for optimization.

2. Continuous Transformation: It is a system in which the outputs actively depend on changing inputs and must be periodically updated. Because of severe time constraints, the entire set of outputs cannot usually be recomputed each time an input changes. Instead, the new output values must be computed incrementally. The transformation can be implemented as a pipeline of functions. The effect of each incremental change in an input value must be propagated through the pipeline. The steps in designing a pipeline for continuous transformation are:

1. Draw a DFD for the system. The input and output actors correspond to data structures whose value change continuously. Data stores within the pipeline show parameters that affect the input-to-output mapping.
2. Define intermediate objects between each pair of successive stages.
3. Differentiate each operation to obtain incremental changes to each stage.
4. Add additional intermediate objects for optimization.

3. Interactive Interface: An interactive interface is a system that is dominated by

interactions between the system and external agents, such as humans, devices or other programs. The external agents are independent of the system, so their inputs cannot be controlled. Interactive interfaces are dominated by the dynamic model. The steps in designing an interactive interface are:

1. Isolate the objects that form the interface from the objects that define the semantics of the application.
2. Use predefined objects to interact with external agents, if possible.
3. Use the dynamic model as the structure of the program.
4. Isolate physical events from logical events.
5. Fully specify the application functions that are invoked by the interface. Make sure that the information to implement them is present.

4. Dynamic Simulation: A dynamic simulation models or tracks objects in the real world. The steps in designing a dynamic simulation are:

1. Identify actors, real-world objects from the object model. The actors have attributes that are periodically updated. The actors have attributes that are periodically updated.
2. Identify discrete events. Discrete events correspond to discrete interactions with the object. Discrete events can be implemented as operations on the object.
3. Identify continuous dependencies.
4. Generally a simulation is driven by a timing loop at a fine time scale. Discrete events between objects can often be exchanged as part of the timing loop.

5. Transaction Manager: A transaction manager is a database system whose main function is to store and access information. The information comes from the application domain. Most transaction managers must deal with multiple users and concurrency. The steps in designing a transaction management system are:

1. Map the object model directly into a database.
2. Determine the units of concurrency. Introduce new classes as needed.
3. Determine the unit of transaction.

4. Design concurrency control for transactions.