**4.5.2 General Components of OO Analysis model**

The key components of an OOA model are

* **Static view of semantic classes.** Requirements are assessed and classes are extracted (and represented) as part of the analysis model. These classes persist throughout the life of the application and are derived based on the semantics of the customer requirements.
* **Static view of attributes.** Every class must be explicitly described. The attributes associated with the class provide a description of the class, as well as a first indication of the operations that are relevant to the class.
* **Static view of relationships.** Objects are "connected" to one another in a variety of ways. The analysis model must represent these relationships so that operations (that affect these connections) can be identified and the design of a messaging approach can be accomplished.
* **Static view of behaviors.** The relationships just noted define a set of behaviors that accommodate the usage scenario (use-cases) of the system. These behaviors are implemented by defining a sequence of operations that achieve them.
* **Dynamic view of communication.** Objects must communicate with one another and do so based on a series of events that cause transition from one state of a system to another
* **Dynamic view of control and time.** The nature and timing of events that cause transitions among states must be described.

**4.5.3 THE OOA PROCESS**

The OOA process does not begin with a concern for objects. Rather, it begins with an understanding of the manner in which the system will be used-by people, if the system is human-interactive; by machines, if the system is involved in process control; or by other programs, if the system coordinates and controls applications. Once the scenario of usage has been defined, the modeling of the software begins. The sections that follow define a series of techniques that may be used to gather basic customer requirements and then define an analysis model for an object-oriented system.

**1. Use-Cases**

Use-cases are an excellent requirements elicitation tool, regardless of the analysis method that is used, use-cases model the system from the end-user's point of view. Created during requirements elicitation, use-cases should achieve the following objectives:

* To define the functional and operational requirements of the system (product) by defining a scenario of usage that is agreed upon by the end-user and the software engineering team.
* To provide a clear and unambiguous description of how the end-user and the system interact with one another.
* To provide a basis for validation testing.

**2. Class-Responsibility-Collaborator Modeling (CRC)**

Class responsibility- collaborator (CRC) modeling provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. The intent is to develop an organized representation of classes. Responsibilities are the attributes and operations that are relevant for the class. Stated simply, a responsibility is "anything the class knows or does". Collaborators are those classes that are required to provide a class with the information needed to complete a responsibility. In general, collaboration implies either a request for information or a request for some action.

**3. Defining Structures and Hierarchies**

Once classes and objects have been identified using the CRC model, the analyst begins to focus on the structure of the class model and the resultant hierarchies that arise as classes and subclasses emerge. A variety of class diagrams can be created. Generalization/specialization class structures can be created for identified classes.

**4. Defining Subjects and Subsystems**

An analysis model for a complex application may have hundreds of classes and dozens of structures. For this reason, it is necessary to define a concise representation that is a digest of the CRC and structure models just described. When a group of all classes collaborate among themselves to accomplish a set of cohesive responsibilities, they are often referred to as subsystem or packages. Subsystems or packages are abstractions that provide a reference or pointer to more detail in the analysis model. When viewed from the outside, a subsystem can be treated as a black box that contains a set of responsibilities and that has its own (outside) collaborators. A subsystem implements one or more contracts with its outside collaborators. A contract is a specific list of requests that collaborators can make of the subsystem.

**5. Software Design**

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and specified, software design is the first of three technical activities (design, code generation, and test) that are required to build and verify the software. Each activity transforms information in a manner that ultimately results in validated computer software.

Each of the elements of the analysis model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in Figure 5.1. Software requirements, manifested by the data, functional, and behavioral models, feed the design task. Using one of a number of design methods, the design task produces a data design, an architectural design, an interface design, and a component design.

The data design transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity. Part of data design may occur in conjunction with the design of software architecture. More detailed data design occurs as each software component is designed.

The *architectural* design defines the relationship between major structural elements of the software, the "design patterns" that can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which architectural design patterns can be applied. The architectural design representation— the framework of a. computer-based system—can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis model.

The *interface* design describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, data and control flow diagrams provide much of the information re required for interface design.

The *component-level* design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the PSPEC, CSPEC, and ST'D serve as the basis for component design.



Figure 5.1 Translating of Analysts Model into Software Design.

The importance of software design can be stated with a single word—quality. Design is the place where quality is fostered in software engineering. Design provides us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support steps that follow. Without design, we risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many money have already been spent.