OBJECT Oriented Analysis and Design for Real-time Systems using the Unified Modeling Language

Firmino dos Santos Filho,
Brazil
totum@totumcontinens.com.br

Abstract
The Unified Modeling Language (UML) is a language for expressing the constructs and relationships of complex systems. UML is more complete than other methods in its support for modeling complex systems and is particularly suited for including real-time embedded systems. Many people use templates derived from IEEE Standard 830-1998, “IEEE Recommended Practice for Software Requirements Specifications” (IEEE 1998). This paper describes the main artifacts to be created during Requirements and Analysis phases in order to design a Real-time System.

Index
Abstract .................................................................................................................. 1
Index ...................................................................................................................... 1
1. Requirements ................................................................................................... 2
  1.1. Functional (textual) description of the system ............................................. 2
  1.2. Context Diagrams (optional) ......................................................................... 2
  1.3. Use Cases ..................................................................................................... 2
  1.4. Scenarios ........................................................................................................ 2
  1.5. Sequence Diagrams ...................................................................................... 2
  1.6. Collaboration Diagrams ............................................................................. 2
2. Analysis: Defining the Object Structure ......................................................... 2
  2.1. Steps in Core Object-Oriented Analysis .................................................... 3
  2.2. Key Strategies for Object Identification ................................................... 3
  2.3. Identifying Object Associations .................................................................. 4
  2.4. Object Attributes ....................................................................................... 5
  2.5. Class Utilities .............................................................................................. 5
  2.6. Verifying the Problem Statement ............................................................... 5
  2.7. Discovering Candidate Classes ............................................................... 5
  2.8. Class Diagrams .......................................................................................... 5
  2.9. Defining Class Relationships and Associations ......................................... 5
  2.10. Associations .............................................................................................. 6
  2.11. Aggregation and Composition .................................................................. 6
  2.12. Associative Classes .................................................................................. 6
  2.13. Generalization Relationships ................................................................... 6
3. References ......................................................................................................... 6
1. Requirements

1.1. **Functional (textual) description of the system.**
   - The software requirements specification is also known as the functional specification, requirements agreement, and system specification. The SRS precisely states the functions and capabilities that a software system must provide and the constraints that it must respect. The SRS is the basis for all subsequent project planning, design, and coding, as well as the foundation for system testing and user documentation. The SRS should describe as completely as possible the intended external, user-visible behaviors of the system. In order to complete the specification it is necessary to use some visual diagrams described below.

1.2. **Context Diagrams (optional)**
   - One of the first steps in understanding what a piece of software needs to do is to study the context in which it must run. Context diagrams treat the system as a composite object and identify the events and messages that transpire between the external actors and the system object. The UML does not explicitly support a context diagram. However the UML object diagram, along with the appropriate stereotypes on messages and objects, serves this purpose nicely.

1.3. **Use Cases**
   - Use Cases are a functional decomposition of the system’s systemic behavior without regard to the system’s internal structure.

1.4. **Scenarios**
   - Scenarios are specific instances of use cases.
   - Two primary scenario models exist: sequence and collaboration diagrams.
   - The complete set of interesting scenarios defines the external behavior of the system.
   - Scenarios are modeled using sequence and collaboration diagrams.

1.5. **Sequence Diagrams**
   - Sequence Diagrams show the sequence of messages between objects.

1.6. **Collaboration Diagrams**
   - Collaboration diagrams focus on the static structure of the collaborating objects. The object structure is clearer than in the sequence diagram.
   - The components of collaboration diagram are:
     - Named objects
     - Relationships joining the objects that exchange messages.
     - Messages with
       - Sequence numbers
       - Identifiers
       - Message direction.

2. Analysis: Defining the Object Structure

Once the system’s external environmental is defined, the analyst must identify the key objects and classes and their relationships within the system itself.
2.1. **Steps in Core Object-Oriented Analysis**

2.1.1. Identify Objects

2.1.2. Identify Object Associations

2.1.3. Group Objects into Classes

2.1.4. Identify and Classify Class Relationships

2.1.5. Identify Class Behaviors

2.1.6. Group Classes into Domains

2.1.7. Validate Classes and Objects

Note: Performing the steps in a different order may be better for some problems or some analysts.

2.2. **Key Strategies for Object Identification**

- Objects of interest
- Attributes of object

2.2.1. Underline the noun

- Try to find the most interesting objects
- The result also includes a couple of attributes

2.2.2. Identify active objects (look for the most fundamental ones)

Active objects are normally implemented as the root composite object of a thread.

- Produce or control actions
- Produce or analyze data
- Provide interfaces to people or devices
- Store information
- Provide services to people or devices
- Contain other types of fundamental objects
- Are transactions of device or person interaction

2.2.3. Identify services (Passive Contributors)

They provide passive control, data storage or both. It provides a service to the active objects. They provide services to client objects.

2.2.4. Identify real world items

Object-oriented systems often need to model the information or behavior of real-world objects even though they are not part of the system per se. If the system manipulates information about these things, then the system should model them as objects.

2.2.5. Identify physical devices (sensor and actuators)

When device information and state must be maintained, the devices may be modeled as objects to told the information about their operational status.
2.2.6. **Identify key concepts**
Key concepts are important abstractions within the domain that have interesting attributes and behaviors.

2.2.7. **Identify transactions**
Transactions are objects arising from the interactions of other objects. One example of transactions are alarms and (reliable) messages. Alarms must persist as long as the dangerous condition is true or until explicitly handled. Reliable message transfer requires that a message persists at the site of the sender until an explicit acknowledgment is received.

2.2.8. **Identify persistent information**
Persistent information typically is held within passive objects such as stacks, queues, trees or databases. Possible persistent information objects: task plans, errors, alarms, hours of operation, security access, service information.

2.2.9. **Identify visual elements**
Visual elements used to convey information to the user are objects within the user interface domain (UI).

2.2.10. **Identify control elements**
Control elements are entities that control other objects. These are specific types of active objects. Some objects, called composites, may be objects such as: PID control loops, Fuzzy logic inference engines, Expert system inference engines, Neural network simulators. Some control elements are physical interface devices that allow users to enter commands.

2.2.11. **Apply scenarios**
The application of use case scenarios is another strategy to identify missing objects. When "you can’t get there from here" occurs, often it identifies one or more missing objects.

2.3. **Identifying Object Associations**
A line drawn between two objects represents a link (instance of an association) between those objects supporting the transmission of a message from one to the other.

2.3.1. **Object Association Strategies**

2.3.1.1. **Identify messages**
Each message implies an association between the participating objects.

2.3.1.2. **Identify message sources**
The sensors that detect information or events and the creators of information or events are all message sources.

2.3.1.3. **Identify message storage depots**

2.3.1.4. **Identify message handlers**
Some objects centralize message dispatching and handling.

2.3.1.5. **Apply scenarios**
Walk through scenarios using the identified objects.
2.4. **Object Attributes**

The UML defines an attribute to be “a named property of a type”. Attributes are the data portion of an object.

2.4.1. **Most important attributes of objects**:
- What information defines the object?
- What information do the object’s operations act upon?
- From the object’s viewpoint, “What do I know?”
- Are the identified attributes rich in either structure or behavior? If so, they are probably objects rather than attributes.
- What are the responsibilities of the object? What information is necessary to fulfill these responsibilities?

2.5. **Class Utilities**

The UML calls traditional functions and procedures *class utilities* and shows them on the class diagram as contained within a class with the stereotype «utility».

On What data does the function operate? If an object encapsulates this data, then put the function within the same object.

2.6. **Verifying the Problem Statement**

Problem statements are the best place from which to start object-oriented analysis.

2.6.1. **Questions that can be addressed to the problem statement (or the domain expert) to elaborate the problem statement**:
- What is the purpose for each object in the environment? How does it interact with the system? Does it act alone or must it work closely with other objects?
- How does the system interact with the user?
- What are the performance constraints on the system?
- What functionality does the system provide?
- Must the system maintain any persistent information?
- Must the system react to any external events? What are the performance requirements of such reactions?

2.7. **Discovering Candidate Classes**

Objects that are identical in structure and behavior are said to be of the same class.

2.8. **Class Diagrams**

Class diagrams are the single most important diagrams in object-oriented analysis and design. They show the structure of the system in terms of classes and objects, including how the objects and classes relate to each other. They are similar to object diagrams except they show classes rather than instances. Multiple relationships between a pair of classes can exist and do so for one of two reasons:

- Different associations with the same objects.
- Associations with different objects.

2.9. **Defining Class Relationships and Associations**

2.9.1. **Message Passing in Analysis**

The abstraction for object communication is *message passing*. Objects communicate by sending each other messages. The message passing schema allows analysts to consider object communication in terms of its semantic content without having to deal with the details of
whether the message is passed by a synchronous function call, sending mail to a task, rendezvousing with a semaphore, or sending a bus message to another processor. Many different implementation mechanisms exist for object communication. Analysis ignores these details, focusing instead on the intrinsic abstractions of the domain and their interactions.

2.9.2. Relationship Taxonomy
Object-oriented systems provide several important types of relationships:
- Association
- Aggregation
- Composition
- Generalization.

2.10. Associations
Associations are logically bi-directional unless explicitly constrained. It is rare that a relationship is actually implemented bidirectionally. In practice relationships are usually navigated only in a single direction (are known as client-server associations). Servers are mostly passive or reactive objects, responding to requests on the part of the client.

2.11. Aggregation and Composition
An aggregation is a special type of association that implies logical or physical containment. The larger class is referred to as the owner or whole, and contains the diamond end of the aggregation. The owner is typically responsible for the creation and destruction of the owned class. Composition is a strong form of aggregation. Components normally are shown by actual inclusion of the component class within the composite.

2.12. Associative Classes
In distributed systems, the classic example of an associative class is a message class. An associative class is used when information does not seem to belong to either object in the association or belongs to both equally.

2.13. Generalization Relationships
Generalization is a taxonomic relationship between classes. The class higher in the taxonomic hierarchy can be called the parent, generalized, base or superclass. Derived classes have all the properties of their parents, but may extend and specialize them.

3. References