Object-Oriented Analysis and Design

Object-Oriented Analysis and Design

Analysis is the process of extracting the "needs" of a system - *what* the system must do to satisfy the customer, not *how* the system will be implemented.

Design is the process of moving the analysis results to a detailed software architecture.

Programming means taking the design and producing an implementation in a particular language.

Analysis and design of object-oriented systems tend to merge together. For both we need to answer:

How do you recognize objects initially?

What are their characteristics?

What are the relationships among the objects?

How do they interact?

The Process to Follow:

1. **Identify the Objects (Principal Objects)**
2. **Define Attributes (Supporting Objects)**
3. **Identify Structure (Classification & Assembly)**
4. **Define Services (Messages)**
5. **Model the Processes (Simulate)**

1. Identify the Objects

Where to look:

Look at the *problem space*, text, pictures. Study *requirements* documents.

Interview users of the prospective application and observe them in action to see what they do, who and what they interact with, in what order, and what the outcomes of different actions are.

Get written specific *cases* or *scripts* of typical interactions.

What to look for:

Identify the ***behaviors*** of the system and the ***actions*** that must be taken.

Then simply look for *who* or *what* is has ***responsibility*** for the actions.

Look for the *nouns* in the scripts.

Example types of objects:

screen interface objects (button, window, etc.)

systems (vehicle, motor, storm)

devices (sensor, switch)

events remembered (registration, oil spill, signal)

roles played (clerk, owner)

locations (site, office)

organizational units (county, department)

What to avoid:

Classes with too much responsibility. *These should delegate some of the responsibility to subordinate objects.*

Classes with no responsibility, no behavior. *If it serves no function, it can be eliminated altogether.*

2. Define Attributes

Identify the attributes.

Attach each attribute to the object it really describes. For example, "color" is really an attribute of a "vehicle" object, not of the "vehicle registration" object.

Identify Supporting objects.

Identify attributes that actually represent supporting objects in what will become an assembly structure. Example: A "sailboat" object has a "sail" attribute which is another object in a part-of assembly structure.

Identification of the supporting objects leads to the assembly, or part-of, hierarchy (see below).

Identify Supporting objects *(continued)*.

Repeated values for some attributes: Example: If a "father" object has attributes "child’s name" and "child’s age", but "father" could have many children, it is probably better to make "child" a separate object with attributes "name" and "age", and put a "list of children" attribute on "father".

Single attributes: Example: If "location" is an object with a single attribute "address" and is related to a "store" object, make "address" an attribute of "store" and eliminate "location" as a separate object.

3. Identify Structure

Look for two types of structure:

- Classification Structure

- Assembly Structure

Classification Structure

Also called: *abstraction hierarchy, is-a, kind-of*

Consider each object as a *generalization* ("vehicle"), and then as a *specialization* ("car"). Put the common attributes higher in the structure.

Group objects according to *similarity of behavior*. Consider forming *abstract classes* which never have instances, but simply group common behavior at a high level and are meant to be subclassed.

Experienced object-oriented programmers advise: *"subclass early and often"*.

Assembly Structure

Also called: *part-of, has-a, composition, containership, mechanism*

Supporting objects already identified as attributes constitute an assembly structure. Consider whether an object can be further decomposed into several smaller functional units, each as a separate object.

Build *mechanisms* consisting of objects which *own* other supporting objects. For example, a "car" object actually owns an "engine", some "wheels", etc.

4. Define Services.

Identify the services.

Consider three types of service:

Occur (instance add, delete, and select).

Calculate. These are the most obvious and represent the major behaviors and functions of the object.

Monitor/change state. These include simple reading/writing of instance variables.

Identify the services. *(continued)*

Look for the *verbs* and *actions* in the scripts obtained in step 1. Review the *behaviors* and *responsibilities* determined earlier.

Look for *dependency relationships* between objects, such that a behavior of one object automatically triggers a behavior in another.

Look for *communication* between objects, such as sending or requesting information.

Ask the key question: "*What should an object in this class know how to do?*"

5. Model the Processes.

Finally, you need to determine which objects *initiate activities*, and identify the *sequence of activities*.

Use the scripts from step 1 and *"run" the model* of the application. That is, given the behaviors, objects, and relationships defined thus far, can the scripts be enacted?

Specify the *life cycles* of objects, and their status at different parts of the cycle. When are instances created and when are they no longer needed?

Iterate through steps 1-5 until you get it right!

References

1. *Object-Oriented Analysis,* by Peter Coad and Edward Yourdon, Prentice-Hall, 1990.

2. *Object-Oriented Design*, by Peter Coad and Edward Yourdon, Prentice-Hall, 1991.

3. *Object-Oriented Design: with Applications*, by Grady Booch, Benjamin-Cummins, 1991.

4. "Objects - Born and Bred", by Elizabeth Gibson, *Byte*, October 1990, pp. 245-254.

5. *Objective-C : Object-Oriented Programming Techniques*, by Lewis J. Pinson & Richard S. Wiener, Addison-Wesley, 1991. Chapter 2 is particularly good on object-oriented design.

6. *An Introduction to Object-Oriented Programming*, by Timothy Budd, Addison-Wesley, 1991. See Chapter 2 on responsibility-driven design.