Patterns of Object-Oriented Design

Joseph Bergin

You have a project to do using object technology and are wondering about how to go about designing the class structure.

Find the Candidate Objects First

Your first task is to find those parts of the system that can best be modeled by objects. Brainstorm to do this without being overly conservative. If it might be an object, list it. For example, in Mancala (see Appendix) the objects might be the board, the pits, the mancalas, the players, the pebbles, a rule enforcer, a scorekeeper.

Try the Nouns

One way to find many of the objects in a system is to read or develop a description of it and then list the important nouns in that description. Nouns name things and objects model things. Apply this to the description of Mancala and see what you come up with. Do the same for the Robot problem.

Model the World

In an initial design try to model the problem directly. In other words, make the program, and in particular the objects, as close to the real-world system as possible. An object in the software system should have an exact counterpart in the world being modeled. This may not be your final design, but it is a good place to start.

Look for Behavior

Next we will examine the candidate objects in more detail and select some of them for modeling. Others may be dropped from consideration at this stage. For potential objects to be retained at this stage we must determine what kind of behavior each of them will exhibit. Behavior means activity. It also means having information that other objects may need. In this case the behavior is providing that information. In Mancala, the players play the game, for example. A pit might "know" how many pebbles it contains and it might be asked by another object like a player how many it holds.

List the behavior of each object. For example, in Mancala the player has behavior "take turn."

If you can’t find any behavior for one of your candidate objects discard it from further consideration.

Think Client-Server

You are trying to discover behavior of a set of candidate objects.
In the Client-Server pattern each object in the system can act as a client to another which acts as server. The client initiates a transaction by sending a message to the server requesting some service. Perhaps this service is a piece of information, and perhaps it is an action that the client would like performed. The server then carries out the request and finally sends any needed information back to the client. In Mancala, the player could be a client to one of the pits, asking it to distribute its pebbles to the neighboring pits according to the rules of the game.

To use this pattern, think about which of your candidate objects needs information or services from which other objects. Then formulate the performance of that service as behavior of the server object.

**Name the Classes**

You have a set of final candidate objects. Some of your objects are of the same kind and some are different from the others. They are "different" if their behavior is different. Name classes for each of the different kinds of objects that you have.

**Distribute the Behavior**

The behaviors that you discovered for your objects now become public methods of the classes that you have just named. Name these methods appropriately. Behaviors (methods) that return information but don't otherwise change things can have nouns for names. The noun should be a good name for the information returned. Behaviors that do things but don't return information should have verbs for names. These names describe the action. Behaviors that do both, should perhaps be redesigned, creating two parts, one which performs the action and the other which returns information (Command-query separation principle).

**Iterate the Design**

Once you have a design and before you attempt to build it, take another look at both the problem and the proposed solution. You may be able to come up with a better design, either by critically examining your current design and modifying it, or by discarding it and using the knowledge gained in its development to build a better one from scratch.

**Factor to Obtain Hierarchies**

One technique that can help modify an existing design to produce a better one is to look for commonalities among the classes/objects in the current design. You can often factor common behaviors into new super classes of the existing classes in a given design. This makes it possible to write code once instead of twice. Look for objects that share some behavior, but have other behaviors that are distinct. For example, in the Robot example, the factories and the rockets do very similar things with the robots. Your initial design perhaps has the factories completely distinct from the rockets. Perhaps a better design would be to have both the Factory and Rocket classes based on a common super class RobotTerminal. The common things that factories and rockets do to robots can be captured as behavior of this class.
Appendix

Example 1 Mancala (from *The Object is Computer Science*)

Mancala is an African game of strategy that is played with pits and pebbles. There are two players. The board has two rows of six pits each, plus a larger pit at each end; the Mancalas. The goal of the game is to collect the most pebbles in your own Mancala. The game starts with four (or five, or six) pebbles in each of the regular pits. Each player has a row of smaller pits in front of them and the Mancala to that player's right. On a turn a player picks up all the stones in one of their own pits and distributes them into consecutive pits moving to the right (counterclockwise), including that player's own Mancala, but not the opponents. If a player's last pebble lands in her/his Mancala, she/he gets another turn. If your last pebble lands in one of your own empty pits then that stone and all of the stones in the opponent's opposite pit if any are captured and placed in your Mancala. The game ends when one player has no pebbles to play. The other player then automatically captures all pebbles remaining on their own side. The player with the most pebbles wins. Other variations of the game are possible. You might try to let the computer play one side and develop a strategy for the computer to play.

Example 2 Robota and Rockets (Ed Epp)

Several robots are being used to shuttle boxes of various kinds from a warehouse (or possibly several warehouses) to a rocket. When a given rocket is full it blasts off and another rocket takes its place and the robots fill it. This continues for as long as you like. The robots move simultaneously (multiple threads of execution) and some of the robots can only handle boxes of a certain kind, or perhaps boxes of a few different kinds but not others. The robots move at different speeds, sometimes dependent on the type of box carried. When unladen, a robot can move a bit faster than when carrying a box. The robots need to negotiate at the warehouse to obtain a box of the correct kind. If none is available, they must wait. They must also wait for service at the warehouse and at the rocket if one or the other is busy with other robots.