revised December 20, 2019

Object Oriented Design

David Casperson

Document Status This version of this document is a DRAFT 2020 project document. Do not (yet!) rely on it not changing.

Overview: Object oriented analysis and design is the process by which we can start from the statement of a problem and arrive at a probable plan for the implementation of object-oriented software to solve the problem.

There are many and detailed methodologies for doing so. What is proposed below, by contrast, is reasonably straight-forward, and is likely to suffice for a problem of the size of the Cpsc 101 project.

Object oriented philosophy: In large (and even small!) software projects, the problem specifications change with time. At the same time, programming experts insist that we should be able to create software libraries that can be reused.

How are these two statements to be reconciled? One useful observation is that problem statements and design requirements evolve much more rapidly than the real world. For instance, there have been several versions of Microsoft Word™ over the past twenty-five years, yet our knowledge of fonts and typographical practice has changed very little in that period.

This leads to the conclusion that the software that we produce should use real-world concepts in its design.

Problem oriented language: In particular the choice of classes and methods and objects should reflect the language of the problem statement. The following procedures give one way to do this:

List of nouns: In order to arrive at a possible list of classes, read through the problem statement and find all of the nouns and noun phrases. Strike from this list those nouns that clearly have nothing to do with the problem to be solved (for instance, "air", "patent", and "handout" all appear in the Score 4 specfication, but are unlikely candidate for classes.) If in doubt, keep the noun!

List of facts: Now re-read the problem statement for the facts contained therein, and for each noun come up with a list of related facts. For instance, for peg, one has at least:

- is a thin metal spike
- there are 16
- arranged in a 4×4 grid
- beads slide to the bottom
- can contain 4 beads.

Some of these facts may later turn out to be irrelevant; try to avoid early judgment. For instance, the fact that a peg is a thin metal spike is likely to be irrelevant to an Ascii-based C++-program, but might be relevant to a robotic implementation or a high-quality graphics version.

Paragraph descriptions: For each noun, write a short paragraph that coherently combines the list of facts found above. Remember that good paragraphs contain a topic sentence!

At this point we are ready to shift to a slightly more programming-oriented frame of mind. The list of nouns are likely candidates for the classes of our program, and their corresponding descriptive paragraphs are likely JavaDoc comments. We know need to find likely methods for these classes.

It is important to remember that at this point the design should concentrate on *what*, not *how*.

Attributes: The attributes of an object help describe what distinguish it from other objects in the same class and may limit the behaviours that object can engage in. For instance, one attribute of a `Peg` may be how many beads are on it — and a `Peg` with four beads should refuse to accept more beads.

More generally, attributes tell us about the state of an object, or give us access to its subobjects.[1]

For each class, come up with a list of attributes for that object. Overdesign here. It may be true that `.isFull()` is the same as `.height()==4`, but it is better to describe both attributes at this point.

Remember that your goal here is to describe *what* rather than *how*. In programming terms, you are describing the signatures and return types of public methods, not their implementation, and not the corresponding private fields.

Behaviours: The behaviours of an object typically result in changes in its state reflected in changes in its attributes. For instance an `.addBead(Bead b)` behaviour of a `Peg` changes the state of the peg. Some behaviours might instead result in changing the state of another object. For instance, a `.printOn(Writer w)` behaviour is unlikely to change the state of the object itself, but will change the state of the object `w`.

In programmatic terms, behaviours likely correspond to the `void` methods and constructors of a class.

For each class, come up with a list of behaviours for that object.

Collaborations: A collaboration is an interaction between two or more objects, especially where those objects are not already related by aggregation. Thus `board.getPeg(0,2)` is usually conceived of as finding an attribute of the board, but `referee.tellTheComputerOpponentToRestart(computerOpponent)` is definitely a collaboration.

Find all of the collaborations that might exist, and for each class, note what the possible collaborations are.

Overdesign! Design work may seem tedious. Frequently computer science students seem reluctant to find classes, attributes, and behaviours in the problem statement. However, it is better to overdesign. If you later decide not to implement your design you haven't invested a large amount of effort. You are much more likely to make mistakes adding to your design later than you are removing from it.

[1] First-year programming texts that are trying to explain the notion of encapsulation sometimes overstate the need to keep things `private`. If a `Player` has access to a board and wants to play a `Bead` on its `A3` peg, the player needs to be able to `board.getPeg(0,2)`, regardless of whether the returned object is officially private or not.

Design Checklist:

- *Can the objects find one another?*

  Another way of phrasing this question is "do your objects have enough attributes describing their physical and logical relationships to other objects?" If a board has pegs, do the pegs know what board they belong to? If a player has access to a board, can the player get access to the pegs of the board? to the colour of the beads on the pegs on the board?

- *Are two words being used to mean the same thing?*

  Do you have both `Peg` and `Spike`? If so, is there an important difference between them, or are they nearly identical?

- *Is one word being used to mean multiple things?*

  Even more importantly, are you using one word for two distinct concepts? For instance, are you simultaneously using the word "location" to mean a two-dimensional location like `A3` and a three-dimensional location like the location of the third bead on the `A3` peg?

  One of the most important things that your design can accomplish is to establish a one-to-one correspondance between concepts and terminology.

To summarize: this is design, not implementation. Use problem-oriented language, not programming-oriented language. Describe *what* not *how*.