OOP Design, Revisted

Much of this material taken from Irving, Ch. 5, and from

I. Design Issues, revisited
   A. Object-oriented design focusses on objects & their relationships, rather than on processes
   B. In traditional top-down design, programs are designed around procedures, with tasks being
divided into subtasks. Reworking the design after this is done can require considerable effort
   C. In contrast, object-oriented design can be viewed as an iterative process: after the initial
design of classes and interfaces, individual components can be constructed. Classes can be
split or combined.
   D. Component design can be divided into the following four steps
      1. Find the classes
         Take the textual description (specification) of the problem and identify the nouns as
candidates for classes
      2. Specify class operations
         Identify the verbs in the textual description as candidates for operations. There are the
following general categories of operations:
         a. Foundation operators (constructors, destructors, copy constructors)
         b. Selectors, or accessors. These return the value of data members, but don’t
            change them)
         c. Modifiers, or mutators. These allow changing data members.
         d. Iterators. These process data members in collections of objects.
      3. Specify class dependencies
         a. Inheritance (is-a, e.g. Triangle i-sa Polygon)
         b. Composition (has-a, e.g. Circle has-a Point)
         c. Link (uses-a, e.g. Mailbox uses-a InputStream)
      4. Specify Class interfaces
         Operations become member functions. Data members are specified.
   E. The above four-step process can now be applied to an example [see handout sheets]
II. Rules of Thumb (Taken from Booch)

A. Every class should embody only about 3-5 distinct responsibilities.

B. A single class hierarchy is suitable for only the most simple application.

C. If your system has more than about 50 to 100 classes, you must decompose your system in terms of class categories to manage the growing complexity.

D. The typical class category will consist of a dozen or so classes.

E. 70% of the classes in a system are relatively easy to discover. 25% of the classes emerge during design and implementation. The remaining 5% are often not found until maintenance.

F. For most systems of modest or greater complexity, focus design on a broad sweep, covering some 80% of the breadth and about 20% of its depth.

G. Do not write code unless you absolutely, positively must. Wherever possible employ application builders.

H. Even the most unusual application should be able to steal at least 10% of its implementation from existing simple frameworks, such as for domain-independent data structures. More mature organization should be able to avoid writing about 30% of their whole system from scratch. The most mature organizations - those that have targetted this approach - can avoid writing 60% or more of their applications.

I. 80% of the errors in an object-oriented system will be found in 20% of its classes.

J. Most good descriptions of a class’s responsibilities can be written with approximately a dozen words.

K. Most interesting classes will on average have about a dozen operations.

L. As a rule, good architectures are composed of forests of classes rather than trees of classes, wherein each hierarchy is generally no deeper than 5+/−2, and no wider than 7 +/-2 at each intermediate node.

M. Most individual operations associated with a class can be implemented in a dozen or so lines of code. It is not unusual to find some implementations requiring only one or two lines of code, however something is very wrong if you find an implementation requiring a hundred or more lines of code.

N. It takes about one month for a professional programmer to learn the syntax and semantics of an OOP such as C++. It takes about 6-9 months for that same developer to embrace the object paradigm.

O. In user-centric applications, approximately 50% of the code will be associated with the look and feel. Much of this code should come from GUI builders.

P. In most OOP’s, the cost of calling a polymorphic operation is about 1.5 times that of calling a procedure in a non OOP language.

Q. All things being equal, a given Smalltalk application will run about 1.5 to 2 times slower than an equivalent C++ application, and the C++ application will tend to run about the same speed as an equivalent C application.
III. An Example
   A. Textual Description (Specification), taken from Irving, p. 159.

   **Doctor’s Office Scheduling Program**

   We want to create a program that schedules appointments for patients in a doctor’s office. The office has multiple doctors, each of which has a daily schedule divided into 15-minute appointment slots beginning at 8:00 AM and finishing at 6:00 PM. In addition to scheduling appointments, we also want to print out a separate daily schedule for each doctor, listing the time and patient name of each appointment.

   The program will be interactive with all output directed to the screen, except for the doctors’ schedules, which will be written to a file for later printing. A restriction we have placed on the program for the moment is that only one appointment day is to be scheduled for each doctor. (i.e. the schedules for the office is set for one day only, not for a week or any other time period requiring some sort of array of schedules).

   A typical interaction with the system would involve a patient calling in, giving their name, and telling which doctor they prefer. The schedule for that doctor would be brought up by the system, showing available time slots. The patient would then choose a time slot, updating the doctor’s schedule and the patient’s records.

   B. Now perform the four design steps. Consider using a "class description form" to keep track of your findings.
      1. Find the classes
         Circle the nouns in the specification, then identify which should be classes.
      2. Specify class operations
         Underline the verbs in the specification, then identify which classes they belong to.
         You should at this point end up with constructors, accessors, mutators, and possibly iterators.
      3. Specify class dependencies: is-a, has-a, uses
         Graphically draw the classes and label their relationships.
      4. Specify Class interfaces: member functions, data members
         Only create the interfaces, i.e. the ".h" files.

   C. Now compare your design to Ivings.
      1. You should find differences due to:
         a. Functionality or data you neglected to consider
         b. Additional abstractions you used that Irving did not use (i.e. inheritance)
      2. Make a list of the differences and make sure you understand which one is better and why.
Class Description Form
(taken from Horstman, p. 93)

<table>
<thead>
<tr>
<th>Name</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Base class(es)</td>
<td></td>
</tr>
<tr>
<td>Purpose</td>
<td></td>
</tr>
<tr>
<td>States</td>
<td></td>
</tr>
<tr>
<td>Constructors</td>
<td></td>
</tr>
<tr>
<td>Operations</td>
<td></td>
</tr>
<tr>
<td>Mutators</td>
<td></td>
</tr>
<tr>
<td>Accessors</td>
<td></td>
</tr>
<tr>
<td>Fields</td>
<td></td>
</tr>
</tbody>
</table>