A Philosophy and Example of CS-1 Programming Projects

Richard E. Pattis
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
(206) 545-1218
PATTIS@CS.WASHINGTON.EDU

Abstract

This paper presents a philosophy underlying CS-1 programming projects, and illustrates this philosophy with a concrete example. Integral to the philosophy is the use of Ada packages (or Modula-2 modules, or Pascal units) and the stepwise-enhancement programming method. The example project specifies a simple program that controls a simulated cardioverter-defibrillator; the first appendix shows a program that meets these specifications. The packages and programs in this paper are written in Ada (the programming language that we use at the University of Washington in our introductory programming courses), but they could be easily transliterated into any language that included the package/module/unit feature.

1 Project Philosophy

In the beginning of a programming course, students must learn the syntax and semantics of a programming language; they must also learn how to build programs, given their specifications, using the language features studied. Programming projects ensure that students achieve both goals. Certainly, such projects must meet the criterion that they require the student to exercise his/her knowledge of the relevant language features and programming method. But well chosen projects can also instruct students about how computers are used in the world outside the classroom, and stimulate them to think about the ramifications of computerization in society.

In my CS-1 courses, I try to assign real-world programming projects (suitably simplified) as early as possible. Interesting applications of computers are reported daily in the media: I frequently find inspiration for projects in newspapers, popular magazines, advertising, and small blurbs in technical magazines. In fact, I often supply such articles from these sources to "prove" to my students the usefulness of the programming projects that I assign. The evidence is all apocryphal, but I believe that students are more motivated and enthusiastic about writing programs whose significance and usefulness they can plainly understand. Here is a summary of one such project, which involves programming a cardioverter-defibrillator (a complete specification for this program appears in Section 3).

Summary: It has been feasible (since the early 1980s) to implant a small device into the chest cavity of a chronic heart patient who is expected to suffer a life-threatening cardiac irregularity (arrhythmia) during the next few months or years. This device, called a cardioverter-defibrillator, includes a microprocessor whose program continually monitors the rhythm of the patient's heart beat; if the program identifies a grossly abnormal heart rhythm, it automatically directs the device to deliver a powerful electrical shock to the heart in an attempt to restore a normal rhythm. This project requires writing a program that controls such a device by monitoring the patient's heart and deciding if and when it is necessary to deliver a shock.

I must simplify the actual specifications for such real-world programs to a level appropriate for beginning programming students. Typically, I accomplish this simplification by paring down the true program specifications, and by providing students with packages that contain useful operations that are beyond their ability to write (see Section 2). Once students have studied types, expressions, statements, and how to read and call subprograms in packages, I can assign them many possible real-world programming projects.

In summary, my programming projects focus on systems, not algorithms (although the systems often include algorithms as constituents). Although there is an acceptable solution to the specified program, the solution to the underlying problem is open-ended and can always be improved with increased knowledge about the problem domain (as is the case in most real-world programs). By their nature, such projects lead to diverse questions — both technical and social — about the problem domain and the efficacy of applying computer solutions to it.

2 Using Packages in Projects

As discussed above, one way to simplify a programming project is to provide students with packages that contain useful operations that are beyond their ability to write. It is often much easier to describe the semantics (for example, by pre- and postconditions) of some complex-to-perform operation than to implement the operation (especially in the case of complicated I/O operations, which are described in more detail below). Before I ask my students to write a complete program, I teach them to read simple package specifications (mostly containing subprograms) and to call the subprograms supplied by these packages in their programs.

This version of a "procedures first" approach is different from the one commonly taught by introductory programming texts. Instead of teaching my students to write subprograms early (before they have learned the control structures necessary to write useful subprograms), I teach my students how to (1) read package specifications, (2) access such packages from their programs, and (3) call any subprograms specified in these packages. This "call before write" approach has two main advantages: it allows students to write more interesting programs early in the course and it familiarizes them with the process of writing programs that call subprograms; so it is more natural for them to continue writing well structured programs after they learn how to write their own subprograms.

The "call before write" approach requires the linguistic ability to cleanly separate a subprogram's specification from its implementation. Ada and Modula-2 are prime examples of languages designed to embody this ability; some Pascal implementations also extend the standard language with "units" to achieve this goal. The ability to build on the work of other programmers is a crucial (but often ignored) part of an introductory programming curriculum; the package/module/unit mechanism is one language feature that allows students to gain this ability easily. When this perspective is taught in a CS-1 course, the more general use of "Abstract Data Types" or "Software Components" in a CS-2 course follows naturally.

Many real-world programs, such as the cardioverter-defibrillator described in this paper, are actually embedded systems. Their interface to the real-world is supplied by a collection of sensor and effector subprograms that control devices external to the computer. An appropriately equipped laboratory could include actual sensors and effectors that are interfaced to a microcomputer. My programming projects use interfaces to simulated devices: the students use a package that contains sensor and effector subprograms that provide a simple interface to the code that simulates the device. Note that the specification of the package is fixed, whether its implementation is connected to a real device or to code that simulates the device. So a simulated implementation can easily be replaced by the real thing; even if real devices are available, students can use the packages that simulate the device to test and debug their programs more effectively.

In general, all I/O is technology dependent. Language designers, in an attempt to insulate their languages from such technological dependencies, have omitted I/O operations. Instead, programmers use a more general mechanism — packages — to specify and implement whatever I/O operations they find useful; the language designers often define standard packages for interfacing to terminals and the file system (on top of which programmers can build their own, more convenient I/O abstractions). This approach meshes nicely with the discussion of embedded systems, where subprograms in packages can specify and implement operations affecting special I/O devices.

In fact, by relegating I/O operations to packages, students can be easily motivated to learn how to read packages and call their subprograms: because they must acquire these skills to perform I/O from their programs. Certainly describing the semantics of such I/O subprograms is not any more difficult than describing the semantics of built-in I/O subprograms — and the syntax and semantics of using packages is quite simple. Therefore, in languages where I/O is implemented with this advanced language feature, instructors are provided a context in which to teach their students how to read and use packages early in their course — a much more important skill to acquire than learning the specific I/O operations of a language.

Finally, besides problem specific packages I also provide my students with various utility packages for primitive types. In the `Integer_Utility` package, for example, I supply an `Inc` procedure (for incrementing variables) and the `Is_Between` and `Is_Opposite_Sign` boolean functions. I also provide my students with a `Trace` package that contains procedures that easily display messages and values on the users terminal (for those students not using the debugger).

3 A Project Specification

This section describes a sample programming project that I assign to my students. In a previous project they have augmented a correct program; in the subsequent project they will learn how to structure their programs with subprograms that they write. By the time that I assign this project, my students have written and hand simulated code fragments containing all the relevant language features.

The actually project that I hand out to my students begins with the summary shown in Section 1, and it continues below with more detailed specifications of the program. In Section 4, following this program specification, is a discussion of a special package that the students use in their programs.

Device Details: The program must somehow interface to a device that can (1) monitor the heart and (2) shock the heart. The sensing operation uses a single electrode that digitizes the electrical activity of the contracting and expanding heart muscle; it always returning an integer value between -10 and +10 inclusive. An example of a normal heart rhythm, and how it is sensed by this device, is shown below in Figure 1.

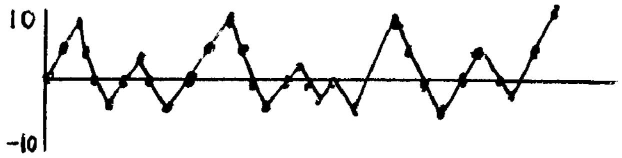

Figure 1: Digitization of a "Normal" Heart Beat

The shocking operation uses an electrode (possibly the same one used for sensing, possibly a different set — it depends on the technology used) to deliver an electrical shock to the heart.

Sense/Shock Algorithm: To determine whether or not to shock the heart, the program will continually sample

```
                         1                     2                       3
N: * 1 2 3 4  5 6 7 8  9 0 1 2 3 4  5 6 7 8  9 0   1 2 3 4  5 6 7 8  9 0 1 2
-------------------------------------------------|------------------------------
C: 0 5 9 5 0 -3 0 3 0 -1 0 5 9 5 0 -3 0 3 0 -1 0  -2 9 5 0 -3 0 3 0 -1 0 5 9 ...
Z: 0          1 2      3 4          5 6      7 8 0 1 2      3 4      5 6     ...
```

Figure 2: Sense/Shock Algorithm Example

a set of 20 values, as one series of data, and count the number of times that consecutive values go from a positive to negative value (or vice-versa). This number is called the "zero crossing count" for that series. The first series of 20 values in the illustration above has a zero crossing count of 8. If after sampling a series of 20 values the zero crossing count lies between 5 and 10 (inclusive) the program should take no special action; this number is considered to be consistent with a normally beating heart. But if the zero crossing count is less than 5, the heart will be assumed to be beating much too slowly; if the zero crossing count is greater than 10, the heart will be assumed to be fibrillating (undergoing rapid, irregular contractions). In both of these cases, the heart is not effectively pumping blood, and the program should apply an electrical shock to the heart, in an attempt to restore a normal rhythm.

Sense/Shock Details: This program should classify 0 as a positive number, even though mathematically zero is classified as neither positive nor negative. When it senses the first value, this program should assume that the previously sensed value (there really isn't one) was 0: note that each zero crossing in computed from two numbers — the currently sensed value and the previously sensed one — so if the first sensed value is negative, the program counts it as a zero crossing. Finally, whenever the program starts another series of sensed values, it should use the last value (from the previous series of values) as the previously sensed value — to determine whether the new series starts with a zero crossing. These details are made concrete in the following example.

Sense/Shock Example: The table above (Figure 2) shows a trace table of the sense/shock algorithm. The `N` row, above the dashed line, shows the number of values sensed, starting with * (before any sensing takes place) and continuing up to 32 (a little over one series). The `C` line shows the current value sensed; as stated above, these values are all between -10 and +10 inclusive. The `Z` line shows the zero crossing count computed so far for each series; this value is incremented whenever the currently sensed value has a sign that is opposite to that of the previously sensed value. After sensing one complete series of values (a | on the dashed line), notice that the zero crossing count is reset to 0; because there was a total of 8 zero crossings for this first series of data, the program should not shock the heart (because this value lies between 5 and 10 inclusive). Finally, notice that the first value in the second series (the 21st data value) causes the program to increment the zero crossing count, because its sign is opposite that of the previous value — the 0 sensed as the last value in the previous series.

Aside: When lecturing about this programming project, at this point I discuss the Therac-25 incident, in which the deaths of 3 persons were discovered to be directly caused by a software malfunction in a cancer radiation-therapy machine. We briefly discuss the benefits of a cardioverter-defibrillator and the various failures possible in such a device, and their consequences (along with the consequences of not using such a device in the first place). For more detailed medical and electrical information about the cardioverter-defibrillator device, and the program that controls it (and the Therac-25 incident), I refer my students to the following articles: [Corcoran 86], [Jacky 89], [Langer 76], and [Mirowski 85].

4 A Package to Monitor/Shock

The program specified above must somehow interface to a device (real or simulated) that can (1) monitor the heart and (2) shock the heart. For this program, I supply my students with a package (placed in a standard library accessible by all students) that simulates such a device. The sensing (monitoring) operation is simulated by reading values from some user-specified file: I provide my students with various test files, containing normal, abnormal, and mixed heart rhythms. The shocking operation is simulated by displaying information on the user's terminal screen (so the programmer can tell when the program decided to shock the heart).

The package also includes a special exception that the **Sense** function raises whenever the file that it is reading contains no more values; I tell my students to terminate their programs whenever this exception is raised. In actual use, the program would continually monitor the heart until it was externally deactivated (or possibly recognized some internal fault and deactivated itself).

My students are allowed to read this package specification, *but not its implementation*. Later in the course, when they are learning to write their own packages, they study this package's implementation as an example, because the students are already familiar with its specification and use. Here, in a condensed form, are the required components for this package. The entire package specification (containing important comments) for this simulated device is shown in Appendix 2.

```
PACKAGE Simulated_Heart_Device IS
  Terminate_Simulation : EXCEPTION;

  FUNCTION  Sense RETURN INTEGER;
  PROCEDURE Shock;
END Simulated_Heart_Device;
```

5 Stepwise-Enhancement

The term "stepwise-refinement" describes a variety of methods that we can use to solve programming problems in a series of steps. I teach my students a stepwise-refinement variation that I call "stepwise-enhancement". This method requires students to formulate a plan that alternates stages of synthesis and analysis, ultimately leading to the development of a program that fully meets its required specifications. After generally discussing stepwise-enhancement in this section, I will illustrate how to apply this technique to build a program that meets the cardioverter-defibrillator specifications.

To use the stepwise-enhancement technique, students first must reduce the program specifications to a minimum, concentrating on their main structural features and ignoring all the complicated details that will make the program difficult to write. Then they design, implement, and test (either by hand simulation or by actually compiling, linking and running on the computer) a complete version of the program that meets these simplest specifications. When they are satisfied that this program is well written and correct according to the simplest specifications, they proceed to the next stage, enhancing the specifications to include some of the complicated details that were previously ignored. Once again they design, implement, and test an enhanced version of the program, which meets the enhanced specifications.

The students continue repeating this process — at each stage enhancing the specifications and writing an enhanced program that meets these new specifications — until they have solved the complete problem described in the original specifications. For each of these stages, I recommend that my students should first hand simulate their code; then enter it onto the computer, remove the compilation errors, and finally link and run the code — and then debug the execution and intent errors (if any remain that were not detected during the hand simulation).

Thus, step by step the students enhance the simplified specifications and their programs, until they have written programs meeting the original specifications. At every stage they are making small additions or modifications to an already correct (for the simplified specifications) program. If at any stage they discover that an enhancement is not correct, they typically need to reexamine only that small amount of code that was added or modified from the previously correct version of the program: they can use this fact to focus their attention when debugging the program. At the end of each stage, it is important to have a well written and easily readable program, since this is the code that they will immediately build upon in the subsequent stages of the stepwise enhancement. Therefore it is critical to finish each stage by simplifying and documenting its code as much as possible, before advancing to the next stage of enhancement.

Fundamentally the stepwise-enhancement technique is useful because it is easier to design, implement, and test a series of increasingly more sophisticated complete programs than it is to attempt writing one large program that solves the original problem specifications at the outset; that is, it is easier to solve a series of many small problems than it is to solve one big problem (commonly called "divide and conquer"). This technique also allows students to test their original ideas on how to solve the main features of the problem in a simple program first. They receive feedback, at very short intervals, that tells them whether or not they are on the correct path to a solution program. So, if their initial ideas are incorrect, they can recognize this fact quickly and discard the ideas early in the programming process, without committing a lot of time and effort to pursuing them; such feedback is critical for students who are learning in parallel the language features and how to use these features when writing programs. If their initial ideas are correct, confirming them in a simplified working program will give the students confidence, as they tackle the more complicated details in the specifications. In either case, students are gaining experience by learning more about the problem and its solution program.

As students gain more programming experience, it will become more obvious to them what are the important structural features in specifications and what are the complicated details; as their programming skill increases, they will be able to implement more complicated specifications at the outset, without having to simplify them further. If parts of the specification are unclear, ambiguous, or just difficult to understand, I advise my students to try to delay coding these parts until the later stages of their programs — so that they can continue coding while seeking clarifications to the specification. At the end of each stage, students should have a working program that they can test on the computer to ensure that it correctly solves the problem at that stage (getting confirming feedback from the computer is vital). After they are convinced that the program at this stage is correct, they should simplify and clarify it as much as possible, before proceeding to the next stage. If they do not finish a program, they still should have a running program that solves a simpler problem.

Now let us examine how to apply this technique to plan the stages of the cardiovertor-defibrillator program, whose specifications were discussed earlier in this paper.

6 Building the Example

For the cardioverter-defibrillator specification, I present my students with the following plan as a stepwise-enhancement in four stages: (1) write a complete program that senses (and displays) all the values and terminates correctly; (2) enhance the program to count each series of 20 values that it senses and display a message at the end of each series; (3) enhance the program to compute the zero crossing count for each series; (4) finally, enhance the program to meet the original specifications, shocking the heart when it detects a grossly abnormal rhythm.

The final program that solves this problem is shown in Appendix 1. Because of space limitations, only the last of these four complete programs can be shown; interested readers should contact me for the first three complete programs.

7 Bibliography

Corcoran 86, Medical Electronics, *IEEE Spectrum*, January 1986, pages 82-84.

Jacky, 89, Programmed for Disaster *The Sciences*, Vol 29, No 5, September/October 1989, pages 22-27.

Langer, et. al. 76, Considerations in the development of the automatic implantable defibrillator, *Medical Instrumentation*, Vol 10, No 3, May-June 1976, pages 163-167.

Mirowski 85, The Automatic Implantable Cardioverter-Defibrillator: An Overview, *Journal of the American College of Cardiology (JACC)*, Vol 6, No 2, August 1985, pages 461-466.

Appendix 1: The Complete Program

```
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
--   Cardiac_Controller is a program that shocks a heart when it detects a
-- grossly abnormal heart beat.  It senses/shocks the heart using subprograms
-- contained in the Simulated_Heart_Device package. It operates by continually
-- processing a series of values, counting the number zero crossings that
-- occur during each complete series, and shocking the heart if this number
-- falls outside a specified range.  ZCC abbreviates Zero Crossing Count.
--
-- Richard E. Pattis
-- CS-210, Fall 1988
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------

WITH Integer_Utility, Simulated_Heart_Device;

PROCEDURE Cardiac_Controller is

  PACKAGE IU RENAMES Integer_Utility;
  PACKAGE HD RENAMES Simulated_Heart_Device;

  ZCC_Series_Size : CONSTANT INTEGER := 20;     -- Series to compute each ZCC
  Minimum_OK_ZCC  : CONSTANT INTEGER :=  5;     -- If < heart beating too slow
  Maximum_OK_ZCC  : CONSTANT INTEGER := 10;     -- If > heart in fibrillation

  Sense_Count     : INTEGER := 0;               -- How many in current series
  ZCC             : INTEGER := 0;               -- Zero crossing count so far
  Old_Sense       : INTEGER := 0;               -- Previously sensed value
  New_Sense       : INTEGER;                    -- Currently sensed value


-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-- Sense_Shock terminates when calling HD.Sense raises an exception; otherwise
-- it updates New_Sense, Sense_Count, (possibly) ZCC and Old_Sense during each
-- iteration. After every ZCC_Series_Size iterations, it decides whether to
-- call HD.Shock, and resets Sense_Count and ZCC for the next iteration.
--

BEGIN
  Sense_Shock: LOOP
    --
    New_Sense:= HD.Sense;                       -- TERMINATE: raise exception?
    --

    IU.Inc(Sense_Count);                        -- Process newly sensed value
    IF IU.Is_Opposite_Sign(Old_Sense, New_Sense)
      THEN IU.Inc(ZCC);
    END IF;

    IF Sense_Count = ZCC_Series_Size            -- Sensed a complete series?
      THEN
        IF NOT IU.Is_Between(Minimum_OK_ZCC, ZCC, Maximum_OK_ZCC)
          THEN HD.Shock;
        END IF;
        Sense_Count:= 0;                        -- Reset for next series
        ZCC        := 0;
      END IF;

    Old_Sense:= New_Sense;                      -- Save new value as old one
  END LOOP Sense_Shock;

EXCEPTION
  WHEN HD.Terminate_Simulation => NULL;         -- Terminate gracefully
END Cardiac_Controller;
```

Appendix 2: The Commented Package Specification

```
------------------------------------------------------------------------------
------------------------------------------------------------------------------
-- Class   : Device Interface
--
-- Author  : Richard E. Pattis
--           Department of Computer Science, FR-35
--           University of Washington
--           Seattle, WA   98195
--           Office Phone: (206) 545-1218
--           Computer Account: C2517 on VAX1
--
-- History : 8/30/1988: Operational
--           9/ 3/1989: Bug fixed in Shock (misspelling in announcement)
--
-- Description:
--
--   This package includes two subprograms that provide a simple interface to
-- a simulated device that can sense a heart signal and shock the heart.  The
-- Sense function returns information that it gets from a file (the user of
-- any program using this package is automatically prompted for the name of a
-- file that contains the simulated data; the Terminate_Simulation exception
-- is raised when there is no more to sense); the Shock procedure displays a
-- message on the user's terminal each time that it is called.
------------------------------------------------------------------------------
------------------------------------------------------------------------------

PACKAGE Simulated_Heart_Device IS


  FUNCTION  Sense RETURN INTEGER;       -- Sense the simulated heart beat
  PROCEDURE Shock;                      -- Shock the simulated heart


  Terminate_Simulation : EXCEPTION;     -- Raised by Sense, when no more data


------------------------------------------------------------------------------
------------------------------------------------------------------------------
--
--                              Semantics
--
--
-- FUNCTION Sense RETURN INTEGER;
--   Pre : See initialization below.
--   PreE: There is data to sense in the file; raises Terminate_Simulation.
--   Post: Sense returns the next simulated heart reading; it will always be
--           an INTEGER value between -10 and +10 inclusive
--   Note: Sense skips any value that is not an integer between -10 and +10.
--
--
-- PROCEDURE Shock;
--   Pre : See initialization below.
--   Post: Shock displays the message "Heart shocked after NNN beats." where
--           NNN is replaced by the number of times Sense has been called
--           since the program began running.
--
-- Initialization: The user will automatically be prompted to enter the name
--   of a file that contains the simulated heart data; if the entered file
--   name cannot be found, the user is reprompted for this information.
--
--
------------------------------------------------------------------------------
------------------------------------------------------------------------------

END Simulated_Heart_Device;
```