Programming Patterns and Design Patterns in the Introductory Computer Science Course

Viera K. Proulx
College of Computer Science
Northeastern University
Boston, MA 02115
vkp@ccs.neu.edu

Abstract

We look at the essential thinking skills students need to learn in the introductory computer science course based on object-oriented programming. We create a framework for such a course based on the elementary programming and design patterns. Some of these patterns are known in the pattern community, others enrich the collection. Our goal is to help students focus on mastering reasoning and design skills before the language idiosynchracies muddy the water.

1 Introduction

The first programming course is a major stumbling block for many students interested in computer science. As the languages get more complex, students spend inordinate amount of time learning minutia of language syntax and some semantics. Meanwhile the overall picture of what is essential and how the pieces fit together is lost in a sea of 'stuff'. Seasoned programmers quickly recognize the key elements needed to write parts of the program and are fluent in creating a complex program from small interacting components. Some of these patterns have been identified by the pattern community and are making their way into the introductory curriculum. However, the methods for presenting patterns to the pattern community are not written as tutorials for novices. An instructor interested in using problem solving in the context of patterns often needs to rewrite these patterns.

Partial support for this work has been provided by the National Science Foundation Leadership in Laboratory Development, award #DUE- 9650552 and by the Microsoft Corporation.

In order to use pattern based problem solving in our introductory computer science courses, we looked at all decisions students must make when designing programs. This inspired us to build a collection of programming and design patterns that lead students through all topics typically covered in an introductory CS course. We decided to present the patterns as tutorials that guide students from the general considerations to the actual implementation in the programming language (C++ or Java). In addition, we built a collection of practice problems that can be used to master each of the patterns.

In the first section of this paper we discuss the problems many of our students encounter when trying to write even simplest programs. We show how this experience helped us to notice some of the patterns of thinking and reasoning we as faculty take for granted. We follow by listing the types of patterns we identified and their role in learning programming and design. To illustrate our approach to presenting patterns we describe the pedagogy used in our courses. This includes both the written materials for students and the actual organization of learning time. We conclude with a brief description of our students' experience in a pattern-centric course.

2 Learning To Program: Why Is It So Hard?

One of the most frustrating experiences when teaching introductory course is meeting a bright student who becomes lost when asked to write even the simplest program. The fact that many students can hand-simulate an algorithm and know exactly what needs to happen at each step - yet cannot program this algorithm has been noticed before. But some of the otherwise competent students cannot program independently even the simplest loop. It seems like we see more of these students as the complexity of the programming languages grows and the core programming and design ideas get lost in the mass of supporting code (the downside of object-oriented languages).

Of course, the key ideas of how to write a loop appear in every textbook. So, the problem lies deeper. The program infrastructure is much more complex than it used to be.

Students get lost in the mire and often just exclaim "I do not even know where to start." One of our students commented on the problem of understanding lectures but not being able to solve lab problems as follows: "Concepts taught in the classroom tend to be an introduction. ... Three lines of code on the blackboard merely show the tool to me but that's about all I saw in the overall view." and later commented "It was like saying, here is...hm...some food, this is the food...now make a meal out of it. Maybe to a person learning to cook for a first time, all this may not make all that much sense."

Looking at what we typically do when writing programs, we realize that we have 'canned methods' for deciding what names to use when - and how to use them, we have a standard way of reading the input, of writing conversions between different data representations, to name just a few 'tricks of the trade'. We address all these problems first at the conceptual level. The programming language considerations come into play only once we know what we want to do. By presenting these patterns of thinking and reasoning we hope that students will be able to apply these methods to their own programs.

In the following section we will identify these basic patterns and organize them into several categories and show how they cover the typical CS1 curriculum.

3 Programming Patterns and Design Patterns for Novices

The patterns we present here play different roles in the design and implementation of a program. The categories often represent an evolving thread that is revisited several times during the first year.

3.1 Name Use Patterns

Every programming language uses 'identifiers' to represent different kinds of entities in a program. Yet the use of all identifiers follows a set pattern: declare - define/build - use - destroy. This pattern helps students realize that every identifier will explicitly or implicitly follow this pattern. When a new kind of entity is presented, students anticipate learning about the declaration, definition or build, use, and destruction.

We first list the kind of entities for which identifiers are needed:

- variables of built-in types and string objects (used for echo printing only)
- constants defined by const or enum
- functions
- file names (even though they may look like strings)
- streams (or file control variables in other languages)
- classes and structs
- object instances
- arrays
- pointer variables
- templates
- user defined types

The declare step binds the identifier to a certain kind of entity (data type, function with a given signature, object in a given class). In many cases this step completely determines what kind of operations are allowed for this identifier.

The second step (define, build, initialize) makes the identifier ready to be used with its full set of behaviors. For variables, this is the initialization step. For a function, this is where the implementation is specified. For a class, the two steps often overlap. Some of the member functions are defined inside the class declarations, others can be defined later. Object instance creation immediately invokes a constructor and so the two steps are (almost) inseparable.

In each case, once the define/build step has been completed, the user interface is specified and the identifier is bound to a specific behavior. The use of an identifier then follows this interface.

By introducing the destroy step early, we set the stage for discussing the memory allocation issues, and the scope and lifetime of an object or variable. Of course, often this step is implicit. However, by including it as a part of the naming pattern students become more aware of how the names are managed by the compiler.

Arrays and pointer variables cover a broader pattern: This pattern of indirect naming will be presented later.

3.2 Reading Data Pattern

Considering the amount of time students spend reading data it is a shame that the whole process is not treated as a serious task. In a typical course functions are used to encapsulate even the simplest computation, yet students keep writing from scratch the typical sequence "user prompt - read data - verify - repeat". We believe students should understand three levels of user input and use them correctly in their programs. The three levels are:

- raw input, where we expect only perfect data and perform no error checking (never used in a production program)
- verified input, where we verify that the input conforms to the expected behavior (e.g. is a number if assigned to a numeric variable)
- filtered input, where we also check that internal constraints are satisfied (e.g. age is >0 and < 150)

Our students use IOTools toolkit [8] to perform the verified input. They then learn to write similar utility functions by implementing filtered input. For example, in a program that computes a grade point average (GPA), the course grade must be a valid grade from a given list.

3.3 Read - Process - Write Pattern

To consider this a pattern may seem trivial or restrictive, but our focus is on the general concept. We focus on the fact that any segment of a program may be specified by describing the input that is needed, the process that will transform this input, and the output or resulting data that will be generated. The input here may come from a sensor, file, or mouse manipulation; the output may be sound, graphics, control signal, as well as plain text. Additionally, the input may not come from an input device - but be contained in function arguments, stored as member data of object invoking a method or take other forms. The output again just means the information generated during the process stage and made available to the user of this code segment - be it a function, method, or an entire program.

3.4 Encapsulation Pattern

In these patterns we try to bring into focus the situations where it is helpful to write a utility function or build a toolkit class. The computational part of these patterns usually falls within a different category. Here the focus is on the need to build a utility tool. We identified four instances where this pattern applies - new ones may emerge later:

- unit conversions (meters to miles, etc.)
- geometric scaling
- encapsulate inner loop (part of repetition pattern)
- encapsulate complex condition (part of repetition and selection patterns)

3.5 Repetition Patterns

This is of course well known topic - both as a formal pattern and as concept covered in every CS class [1]. Our variation divides this pattern into four parts:

- counting - including increments other than one
- conditioned repetition (usually a while loop)
- polled loop (busy wait for external event - e.g. mouse click)
- repetition with exits (usually via break or continue statement)

Our approach is to divide repetition into five stages: initial setup, verifying loop condition, loop body, loop condition update, and post-mortem.

We also include here two loop design strategies:

- design loop body first (to make loop understandable and simple)
- remove from loop repeated computations of the same quantities

This organization takes us away from focusing on which loop control statement to use (for vs. while). The focus is on the nature of the problem and the best solution for that situation. It becomes clear that the different loop control statements implement the same algorithm.

3.6 Selection Pattern

This is another well researched and presented pattern. We include it for completeness. Methods for designing conditionals are included in this pattern [2].

3.7 Traversal Patterns

Any time we work with a collection of data, we need to design a process that will look at all the relevant data items one at a time. The purpose of a traversal is to deliver one data item from a specified collection each time the traversal method is called. For example, the Read Data Pattern may use traversal of input stream to deliver the next item. The following four traversal patterns may be introduced in the first course:

- simple linear traversal (vectors, arrays, strings, file input if using counters or indices)
- streamed traversal - typically terminated by the 'end-of-input' indicator
- linked traversal - traversal of a linked structure
- iterator based traversal [9]

The last three patterns are closely related. They differ in how they handle the responsibility for advancing to the next item and recognizing the end of available data.

3.8 Cumulative Result Patterns

These are composite patterns built out of several earlier patterns. Some variations are known in the pattern world, but we believe that the programmer needs to make several independent and interdependent decisions when designing solutions to this type of problems. The goal is to traverse some collection of data, collecting partial information into some accumulator entity and presenting the composite result at the end. We identify the following four separate components a programmer must consider:

- design the accumulator (Name Use Pattern)
- design update operations (Reading Data Pattern, Selection, Conversion, Formatting Output and other patterns)
- select traversal (Traversal Pattern)
- design post-mortem (may use Formatting Output Pattern)

This pattern has two variations. At the basic level only one accumulator is used. A multipurpose variation may include several accumulators that perform different functions (e.g. computing minimum and maximum in one pass) or even an array of accumulators (for example when computing a histogram).

3.9 Conversion Patterns

Multitudes of textbooks include the conversion of temperature data from Fahrenheit to Celsius and vice versa. However, this is just the tip of the iceberg. Conversion permeates all computing and should be identified as such whenever students encounter it. We classify conversion patterns as follows:

- basic scaling (conversion between frames of reference: minutes and seconds, feet and inches, miles and meters as well as real vs. drawing coordinates) [5]
- casting (either automatic of forced; discusses loss of precision or accuracy, impossibility of some conversions
- formatting output (implicit conversion of integers into strings, etc.)
- table lookup based conversion (mapping is determined by a table)
- encapsulated scaling (constructing scaling class and scaling function object[5])

3.10 Indirect Reference Patterns

Indirect naming comes in several forms. Reference arguments in function calls and array references are the first encounters. Later follows the use of explicit pointer variables and the introduction of iterators. However, they key idea here is that there are several identifiers or identifier-like entities that represent several different kinds of data and each of them is bound to a different behavior. It is best illustrated in the context of arrays. Array name represents a location in memory where the array is stored. It is a pointer. Array index is an integer, with all integer arithmetic and relational operator at our disposal - and with no guarantees about the range. The index is a modifier that may affect the actual location referenced by the pointer. Finally, the array item is identified by a composite name (e.g. a[i]) and its behavior is bound to the base array type. Following N. Parlante, we refer to this entity as pointee.

Understanding the Name Use Pattern helps us in explaining the different behavior of the pointer, pointee, and the modifier.

3.11 Other Patterns

This collection of patterns is not yet complete. We did not look at recursion, algorithmic problem solving patterns (e.g. divide and conquer), or data organization patterns. There is a 'class definition pattern' that includes 'constructor building pattern'. There are also patterns for objects, for example function objects, state objects, data container objects, and derived objects [7].

In our curriculum, most of these will be introduced in the second course. The patterns at this higher level are more likely to be recognized and identified in the pattern literature. Our goal was to create a framework for designing simple programs that will help students understand what are the key questions to ask and what considerations need to be made before the design is completed.

4 Pedagogical Considerations

4.1 Our Course Organization

To guide students through the sea of new ideas introduced in the introductory programming based course, we organize the course as follows. Three weekly lectures introduce the key concepts and outline the relevant patterns. Students read the pattern tutorials together with a problem set document that contains several completely solved problems, several unsolved problems, and a short introduction that suggests a problem solving strategy that student should use when trying to solve the problems and highlights the relevant patterns. Problem sets are discussed during weekly recitation sessions conducted in small groups of no more than 20 students. Students are then required to program and run some of the unsolved problems. The other weekly class meeting of the small group is a closed lab where students implement part of a substantial project that illustrates the use of the new concepts in the context of an interesting application of computer science.

4.2 Pattern Tutorials

Each pattern tutorial is presented in a separate document. The format for pattern tutorials is a modified version of the standard used by the pattern community, to achieve our pedagogical goals. In each pattern we use three levels of explanation: introduce a problem, explore the idea and the related concerns, and finally present the solutions in the context of a particular programming language.

The patterns are written to guide a novice with little or no programming experience who needs a more structured guidance in learning how to program. Better students will also benefit from reading the tutorials, just as it helps experienced programmers to read about patterns. The intuitive decisions become more focused, some of the issues that may have been overlooked will come to the forefront, and possible misconceptions will be corrected. In addition, naming the patterns helps in communicating about the program design with other students and instructors.

We describe briefly the **Name Use Pattern** tutorial to illustrate the three levels of explanations we use.

The first section is called *Intent and Motivation*. In the **Name Use Pattern** we explain the need to name each entity that a program will use, the fact that the programming language has fixed rules how this can be done and that one needs to know what kind of entity a particular identifier represents.

The next two sections discuss the problem at the conceptual level, without using any actual program code.

The *Problem Examples* section gives a few examples with only a rudimentary reference to a programming language. For example, we are printing a weather report, so we will need to record name of each city (string object) and the temperature (integer).

The next section, *Problem and Context*, explains the four steps: declare, define/build, use, destroy and what happens in each step as the program is compiled and executed (i.e. a name is entered into a dictionary and space is allocated, space is filled with a value, value is accessed or modified, space is released and the name is deleted from the dictionary).

The next two sections then focus on the implementation of the pattern in a programming language of choice.

The *Required Elements and Structure* section looks at the actual code that can be used to implement the four steps in this pattern and explains the semantics of the relevant statements and directives.

This is followed by a section of *Implementation Examples* that illustrates all four steps in the context of a several simple problems.

The *Summary* section helps student in remembering the key issues needed to apply the pattern and serves as a quick reference.

A section on *Further Explorations* contains references to other related resources and patterns as well as a brief outline of related but more advanced topics, such as scope, lifetime, memory allocation and deallocation.

4.3 Our Experiences

To make sure students actually read the tutorials, they were required to comment in writing on their usefulness and on any problems they encountered. Most of the novice programmers found the tutorials very helpful, though students with more programming experience felt they already knew most of what was discussed. Some of the comments stated "I thought the information in this section was well though out and clearly presented. The main reason for this clarity is that the information was organized very well and also presented in laymen's terms. Another excellent addition was using both verbal description to point out the different steps and properties of a function, and also concrete examples that the reader could follow easily and apply his/her understanding of the definitions", or " This packet gives a student a much better visual understanding than our text book", or "very helpful, covers a lot, whenever I need to know a specific thing about functions it is there. Thank you.", and "I liked how the handout was set up, giving a definition of a term, explaining it, and then giving several examples on how it works and is used." Overall, we felt the course was a success. Students performed better on the midterm exam and seemed more confident than in the past.

The pattern tutorials and the problem sets are available at our web site: http://www.ccs.neu.edu/teaching/EdGroup/

5 Acknowledgements

I want to thank Joe Bergin for introducing me to the pattern community and providing the inspiration for this work. I also want to thank Richard Rasala, for years of conversations and collaboration that helps in turning ideas into reality. I also acknowledge with gratitude the support of the NSF and the Microsoft Corporation.

References

[1] Astrachan, O. and Wallingford, E. (1998) Loop Patterns. Available: http://www.cs.duke.edu/~ola/patterns/plopd/loops.html

[2] Bergin, J. (1999) Patterns for Selection. Available: http://csis.pace.edu/~bergin/patterns/selection.html

[3] Bergin, J. (1997) Ten Pedagogical Patterns for Teaching Computer Science. Available: http://csis.pace.edu/~bergin/PedPat1.2.html.

[4] Bergin, J. (1998) Six Pedagogical Patterns. Available: http://csis.pace.edu/~bergin/fivepedpat.html.

[5] Deek., F. P., Turoff, M., and McHugh, J. A., A Common Model for Problem Solving and Program Development, *IEEE Transactions on Education*, 4 (1999), 331-336.

[6] Fell, H. J., Proulx, V. K., and Rasala, R. Scaling: A Design Pattern in Introductory Computer Science. *ACM SIGCSE Bulletin* 1 (1998), 326-330.

[7] Rasala, R.. Function Objects, Function Templates, and Passage by Behavior in C++. *ACM SIGCSE Bulletin* 1 (1997), 35-38.

[8] Rasala, R. (1999) Toolkits in Freshman Computer Science: A Pedagogical Imperative, *ACM SIGCSE Bulletin* 1 (1999), to appear.

[9] Rasala, R. A Model Tree Iterator Class for Binary Search Trees. *ACM SIGCSE Bulletin*, 1 (1997), 72-76.

[10] Wolz, U., and Koffman, E. simpleIO: A Java Package for Novice Interactive and Graphic Programming, *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE'99 Conference*, (June 1999), 139-142.