Design Issues in CS Education

Richard Rasala
College of Computer Science
Northeastern University, Boston MA 02115, USA
rasala@ccs.neu.edu
http://www.ccs.neu.edu/home/rasala/

The Importance of Design

It is generally accepted that at the heart of computer science education there are certain fundamentals:

- mathematics and theoretical foundations
- the design and analysis of algorithms
- a knowledge of data structures
- a familiarity with important software systems

To this list of educational fundamentals in computer science, we wish to add a fifth item:

- design issues and skills

Under the broad heading of design issues and skills, we include: the design of functions and classes; the design of large systems with many software components; issues of the concrete versus the abstract; encapsulation and communication; software interfaces and protocols; and user interface design.

We believe that these ideas are of vital importance to computer science and to its applications in the real world. Unfortunately, we also believe that these ideas are not sufficiently represented in the current computer science curriculum and that CS educators must increase both the extent and quality of their coverage of design issues.

In this article, we will discuss some issues in teaching design and will focus on areas where problems exist.

The Role of Exploration in Design

Exploration is fundamental to any creative process. Since our understanding of design is less advanced than our understanding of algorithms, there is need for creative effort even in the most basic aspects of design. There is therefore strong justification for allowing students to work at times in a “play mode” in which they explore a problem and simply seek to find a “solution that works”. This mode may be essential when students are learning a new language or software library.

The problem with this mode of learning is that often students produce code that is very poor from a design perspective. For many students, solving the problem is enough however ugly the code or the design. These students simply stop work and move on to the next assignment.

We believe that students must be taught how to examine code from a design perspective and how to build clean components that work together well. The first step in learning to design is to take code that has been written in an exploratory mode and craft it into a well organized set of functions and classes. Our assignments should require students to do this final step and insist that a solution that works is not necessarily an adequate solution.

Faculty must take the lead in setting design standards by the examples they present in class and the code they provide for assignments. It is not enough to use “throwaway code” to illustrate a language feature. The code you hand out to your students illustrates not only the specific point you are teaching but also your attitude towards design. Be sure to model quality design.

Book authors have an even greater responsibility than individual faculty. Unfortunately, textbooks are often filled with poor quality code that may illustrate language features while sending terrible design messages. Authors must look at each block of code presented and ask: “Does this code stand as an example of quality design?”. If the answer is “No” then the code should be improved because long after the students have learned the specific issue at stake they will come back and look at the code as a design example.

Design Fundamentals

The use of abstraction and information hiding has been a central theme in software design since the early 70's when Parnas [5] first articulated the issues. The recent work of Gamma, Helm, Johnson, and Vlissides on design patterns [2] continues to emphasize the importance of abstraction in building systems with design flexibility.

Unfortunately, with all this emphasis on abstraction and information hiding, there has been a tendency to lose sight of the fact that at the base of the software hierarchy there must be a collection of quality concrete data types or classes on which to build. By shining a single spotlight on abstraction, we have lost the pedagogical opportunity to clarify the more subtle issues including the role of concrete data types and the need for varying degrees and kinds of abstraction.

In many elementary textbooks, authors immediately begin discussion of abstract data types (ADTs) with no firm foundation of concrete data types (notice that there is not even an abbreviation for CDTs). No genuine distinction is made between the abstract and the concrete. As a result, many concrete details creep into the interfaces and the implementations of so-called ADTs.

Since understanding the relationship of the concrete to the abstract is so important for the pedagogy of teaching design, we will attempt to clarify some of the key issues.

The Purposes of Encapsulation

Encapsulation puts things into capsules. The most important reason for doing encapsulation is to make a program more comprehensible. By naming data sets or sequences of actions, we can refer to these entities conceptually rather than by reciting their explicit representation. In addition to this basic *organizational role*, encapsulation also serves the following purposes:

- *completeness*: collect all actions that need to be done on a data set
- *safety*: encode safety checks that would tend to be forgotten if left to inline code
- *security*: maintain security mechanisms that control authorized access
- *uniformity*: present a *standard* interface so that objects may be used interchangeably in a larger program
- *parsimony*: present a *limited* interface so that the larger program cannot make detailed assumptions about an object.

Note that these purposes may be applicable to both concrete and abstract data types. There is no reason to view encapsulation as a mechanism useful only for abstract data types. How encapsulation is used depends on the goals of a particular component.

The Role of Concrete Data Types or Classes

Concrete classes should emphasize completeness and safety rather than uniformity and parsimony. A concrete class should make clear what structures it is working on and what algorithms it offers. The design should be comprehensive rather than minimal. The goal should be to build a powerful tool which can support many different abstract purposes.

It is in concrete classes that fundamental algorithmic work must be done. The role of a concrete class is to manage memory, provide access mechanisms, and support algorithmic work. A concrete class is a tool and a tool must be visible to be useful. What is hidden in a concrete class are the safety mechanisms that permit the tool to be used without danger.

The Role of Abstract Data Types or Classes

Abstract classes should emphasize uniformity and parsimony by encapsulating information that needs to be hidden and by providing information that needs to be communicated. The decision about what needs to be hidden or communicated must be made in the context of the particular software that is being designed. Hiding hedges a designer's bets about what implementation details are likely to change. As [2] points out, it is impossible to have infinite design flexibility and allow for all possible changes. Therefore, abstract classes need to be tuned to a particular programming situation and the design areas most likely to undergo change.

This insight has important pedagogical implications. For one thing, it explains why if you examine a dozen textbooks on data structures you will find a dozen variants of an abstract *list class*. Each author has envisioned the use of the list class within the context of the software in the book and chosen a set of abstractions to communicate and a collection of details to hide. Since each book has a different teaching philosophy, it is no wonder that these list abstractions turn out to be different.

As teachers, we should not attempt to present *the* list class but should rather discuss the variety of ways in which a list might be abstracted to meet specific design flexibility goals.

More generally, abstract classes are not the basic unit of reusable software. What is resuable are concrete classes that are complete and relatively open to access and extension. Pedagogically, this means that we must give careful consideration to the design of concrete classes as well as to the building of abstract layers on top of such classes.

The implications of the need to tune abstract classes to the design flexibility goals of a particular piece of software is that abstract classes had better be easy to build. To accomplish this, the member functions in an abstract class must have simple implemenations. Ideally, each member function in an abstract class should be implemented by at most three lines of code that refers to some underlying concrete object: setup, make-concrete-call, cleanup.

Degrees of Abstraction and Practical Issues

Of course, in a practical world (both in industry and in the classroom), we do not always do what is ideal. There is a great temptation to design an abstract interface and then code substantial algorithmic details into the implementation. However, *to the degree that an abstract class does concrete algorithmic work, it ceases to be abstract*. If an abstract class does concrete algorithmic work then it will not be easy to change these implementation details and the benefits of abstraction will be lost. At the same time, the algorithmic work will be hidden by an interface which fails to deliver its full potential to the caller. Hence, when the algorithm is needed again in a different context, the class wrapper may have to be redesigned anyway.

Let's name this phenomenon. We will call a class *quasi-abstract* if it has an abstract interface but has an implementation which directly codes substantial algorithmic details.

Many examples used in texts for abstract classes are really more precisely quasi-abstract classes. This means that the data representation is hidden properly but that the algorithmic work has been done inline rather than being similarly encapsulated. To some degree, such a class is a design compromise and should be recognized as such in a teaching setting. Recognition of this compromise is crucial to dealing with it from a design perspective.

The reason as educators that we tend to drift into quasi-abstract classes is twofold. On one hand, we do not have a robust enough set of concrete classes to build upon. On the other hand, to create a truly abstract class, we may need to add another level of abstraction and the pain of additional names and class definitions is too much to bear.

Certainly, in the context of busy courses, there is sometimes not enough time to do everything in the best possible way. Nevertheless, we should try to set design standards and, when we need to compromise, we should be explicit about what we are doing and why. Teaching is about being aware.

Toolkits and Environments

If we wish to teach design then we must work in an environment that is rich in tools. As Eric Roberts states [10]:

> Students need exposure to modern software development practices. It is no longer acceptable to teach programming by having students write small, self-contained programs from scratch. Even though we cannot predict exactly what toolkits and programming environments students will use when they graduate, it is essential that they learn to work with some modern programming environment that forces them to add small amounts of special-purpose code on top of a large existing infrastructure. Moreover, this experience should come early in the curriculum so that students can apply these techniques in summer internships and their more advanced courses.

In addition to reflecting current practice, there are important pedagogical reasons for students to work within a framework of existing code, tools, and class structures.

The most important reason for asking students to extend or modify existing code is to permit them to see and explore a large base of well designed code. If we provide a partial solution to an assignment, we have the opportunity to model the kind of software design principles we preach. If we provide a set of tools to be used in a number of assignments, we have a chance to show what we mean by quality reusable software. The code we deliver to students is our stake in the ground about what quality design means in practice.

As students progress in the curriculum, they must be exposed to large scale commercial development environments. Although source code is usually not available in such environments, students can still learn by studying the pros and cons of the class libraries provided in the environment. A large scale class library may be used as a set of examples against which the design methodologies in textbooks can be examined. Students can learn about the strengths and weaknesses of both the libraries and the methodologies.

It must be recognized that it will not be easy to have faculty everywhere use framework code, toolkits, and advanced environments. Few faculty have the support structure necessary to build large quantities of code. Those of us who have designed and built such pedagogical code must make every effort to distribute this work to faculty at other institutions. However, due to portability issues, this distribution task is not easy. Therefore, it will take time for good code and lab depositories to be put into place.

The main obstacle to the faculty use of large scale commercial development environments is the time it takes to grasp the class libraries. With documents that are several inches thick, it must be considered a non-trivial task for faculty to learn an environment well enough to use it in teaching. Hopefully, CS units will begin to allow time for their faculty to pursue this learning process.

Design and the Software Project Course

In most institutions, students take a junior/senior project course in which they learn about software design and development in a team setting. Ideally, this course should provide a situation in which the design principles learned in earlier courses are put into practice and new design principles applicable to large scale projects are taught.

Unfortunately, it is often the case that the project course is a negative design experience. The course is often treated as a rite-of-passage in which students are expected to work ridiculously long hours in order to finish a project on time.

Under these circumstances, there is no incentive for students to ponder a design, explore alternatives, and backtrack to make improvements. Just as in the freshman courses, solving the problem is enough however ugly the code or the design. Of course, students know that there is some penalty for poor design but they also know that the biggest penalty is for not having a running program that roughly meets the specs.

We believe that the project course must cease to be a rite-of-passage and instead must become a serious investigation of design issues. Using an incremental approach to building software, students should be encouraged to add class structures and then build working partial solutions every 2-3 days if possible. If students “fall behind” in terms of meeting the most ambitious project goals, they should be permitted to “make up the grade” by explaining in writing what design pathways they explored and why their design is of higher quality because of this exploration. The grades will then depend not only on how much the students did but also on what they learned along the way.

Design is ultimately a human process and, beyond a reasonable level, stress is counter-productive for the creative work that design requires. If we believe that it is possible to have both rapid development and well designed software [3], then we must use the project course to teach students how this can be accomplished. We must be models of best practice and we must not encourage the mistakes so often committed in industry. We need to make the project course a calm, humane, creative, and rational design experience. Our students will then be able to bring to their future work both good methodologies and good attitudes.

Conclusions

In this article, we argue that design is important throughout the computer science curriculum. We believe that faculty must look at everything they teach and ask: “What are the design implications of this material?” and “What must I do and what must my students do to address these implications?”. A strong commitment to teaching design is necessary if we want students to develop an integrated view of the computer science domain.

There are a number of problems in teaching design that we have discussed:

- insufficient focus on design
- design experiences of poor quality
- lack of clarity about fundamental issues.

To begin to address these problems, we wish to make some pedagogical recommendations:

- Consider the design implications of every topic
- Insist that students polish the design aspects of their programs before handing in their work.
- Make the sample software you use a model of design excellence.
- Find good illustrations for both concrete and abstract classes.
- Explore design methodologies in practice by conducting software design experiments.
- Use libraries, toolkits, and frameworks to teach students how to integrate into a larger software design environment.
- Make the software project course a quality design experience not just a rite-of-passage.

Acknowledgements

I wish to acknowledge my collaborators Viera Proulx and Harriet Fell who work with me on the development of curriculum materials and my dean Larry Finkelstein who genuinely cares about the results of this work. I wish to thank my colleagues at Northeastern and my friends at other institutions for many stimulating conversations on design issues. Finally, I wish to thank the NSF for partial support under grant DUE-9650552.

Bibliography

[1] Timothy Budd, Data Structures in C++ Using The Standard Template Library, , Addison-Wesley, Reading, MA, 1998.

[2] Erich Gamma, Richard Helm, Ralph Johnson, & John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[3] Steve McConnell, Rapid Development: Taming Wild Software Schedules, Microsoft Press, Redmond, WA, 1996.

[4] David R. Musser & Atul Saini, STL Tutorial and Reference Guide: C++ Programming With the Standard Template Library, , Addison-Wesley, Reading, MA, 1996.

[5] D. L. Parnas, On the Criteria to Be Used in Decomposing Systems into Modules, Comm ACM, 5(12), 1972, 1053-1058.

[6] Viera K. Proulx, Richard Rasala,& Harriet Fell, Foundations of Computer Science: What are they and how do we teach them?, SIGCSE Bulletin, 28(Barcelona Conference), 1996, 42-48.

[7] Viera K. Proulx & Richard Rasala, On The Future of Computer Science Education, ACM Computing Surveys, 28(4es), December 1996, on-line at www.acm.org.

[8] Richard Rasala, A Model Tree Iterator Class for Binary Search Trees, SIGCSE Bulletin, 29(1), 1997, 72-76.

[9] Richard Rasala, Function Objects, Function Templates, and Passage By Behavior in C++, SIGCSE Bulletin, 29(1), 1997, 35-38.

[10] Eric Roberts, Directions in Computer Science Education, ACM Computing Surveys, 28(4es), December 1996, on-line at www.acm.org.