Design Patterns: An Essential Component of CS Curricula

Owen Astrachan*, Geoffrey Berry, Landon Cox, Garrett Mitchener
Duke University
{ola, gcb, lpc1, garrett}@cs.duke.edu

Abstract

The field of software patterns has seen an explosion in interest in the last three years. Work to date has been on the recognition, cataloging, and finding of patterns with little attention to the *use* of patterns, especially by students and practitioners not well-versed in object-oriented technologies. This project addresses pattern use through the development of several programming and pedagogical frameworks that supply support for using patterns throughout a computer science curriculum. Although we do not claim that patterns are Brooks' silver bullet [10], their use can help cope with the accidental complexity of software development and, we argue, their use is essential for a successful adoption of object-oriented techniques in academic computer science programs. This project addresses practical concerns of the computer science and software engineering communities in using, teaching, and learning patterns. In this paper we argue that patterns are an essential programming and pedagogical tool and report on our work in making them accessible to the educational community.

1 Introduction

One principle tenet of object technology and the cornerstone of academic programming courses in which students use code and libraries supplied by others, is *software reuse*. For large programs and projects, reuse must occur at a level above an individual class or even a class library. This is described in the first Pattern Languages of Program Design Conference proceedings [17].

> Experience over the past two decades, however, indicates that it is difficult to achieve reuse and extensibility via class libraries alone. The basic problem is that the scope of a broadly reusable class is not large enough to significantly reduce the amount of application code that must be developed manually.

This observation has led to the development of program frameworks [42], software architectures [33], and design patterns. Design patterns have their roots in the architectural work of Christopher Alexander [1, 2]. This work was first brought to the attention of the software community in [9] but more forcefully and with great impact in the seminal work [20]. Of course earlier work such as [28, 36, 35] provided foundational material for the adoption of architectural patterns by academics and software practitioners. Work continues to be reported in many conferences and recently in conferences devoted specifically to the use of patterns and pattern languages [17, 39]. Related work can be found in [11, 12, 13, 15, 22]. Supporting views for the use of patterns in software architectures and other related areas are found in [33, 34, 24].

Our project is designed to make *software patterns* accessible and useful in an undergraduate computer science curriculum. In this paper we describe patterns, their potential impact as an integral part of a curriculum, methods for incorporating patterns early and often throughout a sequence of courses, and supporting frameworks we have developed to facilitate the use and understanding of patterns by educators. In Section 2 we define design patterns and idioms at a high level; in Section 3 we describe why patterns are useful but potentially difficult to use; in Section 4 we provide examples of patterns and their successful use; in Section 5 we describe class libraries and expository material for using patterns; in Section 6 we describe a potential rethinking and restructuring of curricula that might be facilitated in part by using patterns; Section 7 discusses related and future work; in Appendix A we describe potential pitfalls in web-based methods of dissemination

2 What is a Pattern?

Alexander defines a pattern as follows: [2]

> A design pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution. The pattern is, in short, at the same time a thing, ..., and the rule which tells us how to create that thing, and when we must create it.

*This work is supported in part by the National Science Foundation grants #DUE-9554910 and #CCR-9702550.

This definition has been extended and modified by the software community to a schema comprised of four parts [20]:

- The **name** which provides a handle and a vocabulary for discussing and using the pattern.
- The **problem** which provides a context in which the pattern is applicable.
- The **solution** which describes the components of the pattern and how they interact including their responsibilities, relationships, and collaborations. The solution is abstract although it is often useful to ground it with a concrete example in a specific programming language.
- The **consequences** of using the pattern include tradeoffs and implications that arise from adopting the solution to the problem in context.

Patterns help extend the vocabulary of students by providing a toolkit of higher-level concepts. The extension of a programmer's vocabulary is particularly important, it allows us (students and educators) to engage in more fruitful design discussions. The *name* of the pattern becomes particularly important in this context. Once a pattern is understood, both conceptually and from useful experience[1], it becomes part of a student's vocabulary and raises the level of discourse considerably. Design patterns transcend particular languages although not, necessarily, language paradigms. The language independence of patterns makes them useful with any object-oriented language. *Idioms* are similar to patterns but are closer to the idea of templates and case studies [30, 32] and are often grounded in a specific programming language. Our project addresses both design patterns and idioms.

Idioms have a long history [36, 30] and are recognized as part of the patterns community [16, 11]. Although the difference between an idiom and a pattern is not always clear, the language specificity of idioms differentiates them from design patterns in most cases. Both kinds of patterns are essential for those new to object technology since object-oriented programming and design will be realized in a specific language.

3 The Need for Patterns

Most introductory textbooks attempt to do service to the process of "problem solving" and "program design". However, the majority of texts are driven by the syntactic details of a specific language rather than by general methods for solving problems and designing programs. Students may come away from courses based on such books with concepts described in words as "*divide and conquer*", and "*top-down design*", but our experience is that students are not adept at designing programs or solving problems after one or two semesters. Instead, we propose that students should be exposed to good designs in an approach similar to that advocated in [3, 8]. Programming and design vocabularies should be increased using idioms and patterns such as those described in this paper and elsewhere [40, 41, 21]. Students should use libraries and frameworks built using a pattern vocabulary and should use these libraries in several related, introductory courses.

The use of patterns for those being introduced to object technology is described in [15]:

> Patterns capture established practices that remain obscure in the broad practice of a given domain. Many patterns have their roots in the work of early adaptors of a new technology or the first architects of a system. Many of these patterns attack problems in subtle ways, which makes it difficult to cast them in the framework of the predominate constructs of the system. For example, when people first learn C++, they learn it in terms of language features: classes, functions, and objects or in terms of object-oriented design principles such as inheritance and polymorphism that lead to good class partitioning. However, certain C++ idioms transcend the language and are best expressed as higher-level or meta concepts. These concepts are captured as patterns when the patterns are expressed as solutions to a problem in context.

3.1 Problems Using Patterns

The strength, purpose, and abstractness of design patterns makes them very accessible to those well-versed in object technology, but less so to those new to the field. This disparity between effectiveness and accessibility is especially evident with students and educators new to the discipline. One goal of this project addresses this disparity, with methods for cataloging and using patterns that make them accessible to students and educators (and software professionals).

The kind of design catalog provided in [20] is not accessible to students, educators, or programmers new to object-oriented programming. The pattern literature has focused on experienced users of object technology, allowing them to find better and more useful ways of organizing concepts and frameworks to facilitate design and reuse of software components. Our project tackles this problem directly by answering concerns voiced in [11] as fundamental to the success of the patterns community in the near future:

> The ability to find (and a prerequisite to use) patterns decreases in proportion to the number of documented patterns. Although the patterns in [20] are divided into three subgroups (creational, structural, behavioral) the patterns are still difficult to use without experience.

[1] Here we literally mean *use*ful, i.e., has been used in a design or program.

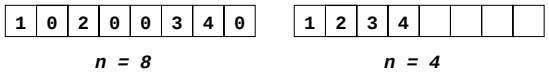
4 Pattern Examples

In this section we describe several patterns and the materials we have developed for incorporating these patterns into our curriculum and for other institutions to use in their curricula. Our philosophy is to test materials ourselves, but to take steps to ensure that they are useful in a wide range of institutions. One of the grants that supports this work supports two collaborating institutions with drastically different student populations including one HBC (Historically Black College), one undergraduate institution, and one research university.

For this project, we have developed several pedagogical frameworks of code, libraries, expository material, and assignments that are accessible at `http://www.cs.duke.edu/csed`. The materials provide resources for educators on creating assignments and using patterns in conjunction with the apprentice style of learning outlined in [3, 8].

4.1 Simple Program Patterns

Although patterns are known primarily through [20] , these patterns address design concerns of object-oriented systems. These patterns are important, and must be covered in a course of study addressing object-oriented programming and design, but simpler patterns can be used in studying programming at the level of loops and conditionals. Using programming patterns as a means of cataloging programming techniques provides a foundation for the introduction of design patterns. In [40], Eugene Wallingford outlines his use of programming patterns for both procedural and object-oriented programming. The patterns and mode of teaching that he proposes are an essential foundation for subsequent work with more complex design patterns. As an example, consider the problem of processing sequential data, e.g., all the values in an array, or the values in a file. When the processing is complex, beginning students often try to use two loops where one loop leads to code that is easier to understand and easier to develop correctly. To see this problem in context, ask students to write code to remove all zeroes from an array, leaving the order of the other elements unchanged, i.e., to change the array on the left below to the array on the right.



In our experience, most students write one loop to process all the array entries, and one loop to try to find and remove zeros. This invariably leads to off-by-one bugs, problems running off the end of the array, and problems with boundary cases (e.g., first zero, last zero). A simple loop with an if statement tracking the last non-zero index leads to short, concise code; see [4]. As another example, consider processing run-length encoded representations of black and white pixel data for rectangular images, e.g., 4 1 2 3 represents `0000100111`. With no guidance, students often try to write nested loops to fill a matrix, rather than a single loop to read the file and populate the matrix. Captured as a pattern we might call this problem *One Loop for Linear Structures* and categorize it briefly as

> Algorithmically, a problem may seem to call for multiple loops to match intuition on how control structures are used to program a solution to the problem, but the data is stored sequentially, e.g., in an array or a file. Programming based on control leads to more problems than programming based on structure.
>
> *Therefore*, use the structure of the data to guide the programmed solution: one loop for sequential data with appropriately guarded conditionals to implement the control.

4.2 Iterators

In our first course we use the pattern identified in [20] as *Iterator*. Iterators solve the problem of providing sequential access to the elements of an aggregate without knowledge of how the aggregate is implemented. We have had great success using iterators of different types in our courses. We use iterators to hide the details of how aggregate data is stored and to build a foundation that we follow through three courses on programming, design, and computer science.

In our first course we use a class to read words from a file using member functions *first*, *isDone*, *next*, and *current*. The code below uses the pattern, realized in a class *WordStreamIterator*, to calculate the average word length in *Hamlet*. Here the aggregate is a collection of words, each word is extracted one-at-a-time using the iterator pattern.

```
int main()
{
  WordStreamIterator iter;
  iter.open("hamlet");
  string word;
  int totalWords = 0, totalLetters = 0;

  for(iter.first();!iter.isDone();iter.next())
  {
     totalWords++;
     totalLetters += current().length();
  }
  cout << "average word length = "
       << double(totalLetters)/totalWords << endl;
}
```

We use the Iterator pattern with the same names for member functions in a class that returns the entries in a directory, either files or subdirectories. Students use this class to practice recursion in a naturally recursive context: e.g., subdirectories of subdirectories as described in [5]. Students are accustomed to the iterator pattern and have little difficulty in applying the functions in a new context. In our second course we migrate to the kind of external pattern described

in [20] that is part of an inheritance hierarchy. Because the pattern is the same, and the member functions have the same names, our students have little difficulty in coping with the new method for implementing the pattern. In contrast, when we've introduced this more advanced method without the context of the first course (e.g., in a graduate course in software design) students had more difficulty. Using the iterator pattern also builds a foundation for studying iterators as used in the C++ Standard Template Library and with the Java *Enumeration* interface.

The iterator pattern by itself is of little value without several classes that provide a context and an application for its use. Although experienced programmers can sometimes see the benefits of a pattern abstractly, students and educators new to object technology must have examples to make the patterns concrete before realizing the power of patterns abstractly. The examples we have developed show the preliminary promise of this line of research. The examples are motivating because they make significant use of computational resources, e.g., solving problems that cannot be solved without the computer.

4.3 An Image Processing Framework

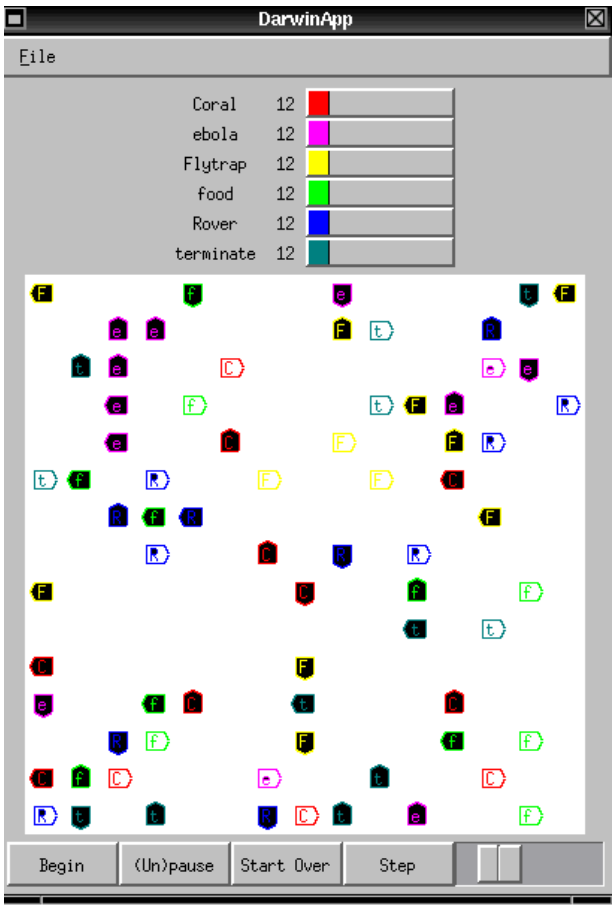
One of the frameworks we have developed uses image processing of as the basis for labs and assignments in introductory programming courses. Students are given partially complete classes that they use to load, display, and manipulate images in several formats, see [6] for details. We use this program in our first course, and in an accelerated course that combines two semesters into one for students with prior experience. Students are asked to implement different parts of the program in different semesters, so that the same framework supplies material for related, but different assignments across semesters.

In our software design course we dissect the version of the program used in earlier courses, and show how several patterns make it more general and more adaptable to use with either a GUI or text UI. In particular, we use the *Command*, *Proxy*, *Composite*, and *Mediator* patterns from [20]. A text-based version using C++ exists for Unix machines, and GUI versions exist for both C++ and Java running on other platforms.

4.4 A Simulation Pattern Framework

A preliminary version of this assignment framework was reported in [7]. For the last two years, we have used the framework successfully in a one-day workshop introducing women and minorities to computer science The program is related to the kind of Artificial Life programs described in [25, 29] and is based on assignment given at Stanford and developed by Nick Parlante.

This framework is called *Darwin* and is based on a two-dimensional grid world inhabited by creatures competing for world domination. Creatures run a program specific to their species and written in *DULL* (Darwin Unstructured Lattice Language). These programs control how and when a creature can hop, turn, see what's in front of it, and, most importantly, infect other creatures. When creature *x* infects creature *y*, *y*'s species changes to *x*'s species. Species programs are constructed so that creatures exhibit specific behaviors (e.g., some species make creatures that are food for other creatures), and students write DULL programs for species to take over the world. A screen shot of the Darwin simulation is shown below.



An example DULL program follows:

```
Flytrap
ifenemy 4
left
go 1
infect
go 1

The flytrap sits in one place and spins.
It infects anything in front of it
Flytraps do well when they clump
```

4.4.1 The Observer Pattern

The Model View Controller (MVC) pattern from Smalltalk [23] is discussed in [20] as *Observer* and is realized explicitly in Java with Observer/Observable interfaces. The Observer pattern is appropriate when a change in

one object requires that other, observing objects update their state to reflect the change. The goal is to decouple the object being observed from the objects that are observing so that they can be reused and varied independently. The solution uses two separate class hierarchies: a model and a view or, in Java, the observable and the observer. The MVC pattern can be viewed as a Smalltalk idiom (supported directly in the language), a design pattern (solving a coupling problem), and an architectural framework (structuring a large system) [15]. These different views can be discussed at different points in a curriculum.

In the Darwin program, each creature is an observable with several views. The world is one observer/view, the bar-gauges showing how many of each species are alive is another view, and an animated view of an executing program is a third view. Students don't write the observers, but typically write the code for a class *Creature* and must update observers when appropriate. Related work [8] described the use of the Observer pattern in a program that showed a graphical example of automatic bowling scoring. One bowler, the observable/model is watched by two separate observers/views: one to show the score automatically, and one to show a graphical simulation of pins being knocked down.

4.4.2 Using Darwin

In this section we show how patterns can improve on existing practice by showing how Darwin was originally implemented in C, how this technique can be implemented in C++ using an idiom, and how a Factory pattern improves on both implementations. The Factory pattern [20] facilitates alternative implementations of a class or abstract data type (ADT). We are using a factory to isolate client code from dependencies on specific class names, i.e., we do not want to require students to use a specific name like *StudentCreature* for the class they implement. Our students do struggle at first with this pattern, but the context shows how patterns help in making a transition to an object-oriented way of thinking. As part of this project we hope to develop material that makes the factory pattern more accessible .

We have modeled our intended use of this assignment after the original Stanford description. The original program was designed to be implemented in C, with pointer-based ADTs for creatures, species, and other types used in building the Darwin program. C supports forward declaration of pointer-based types, e.g., `Creature * c` and code that uses these types, e.g., `takeTurn(Creature * c)` and `getSpecies(Creature * c)`, without needing the concrete implementation of the *Creature* ADT. Students are provided with compiled code for each class so that they begin with an executable program. Each ADT is re-implemented and the new implementation linked with provided implementations of other ADTs to create a (hopefully) working program. Students learn the benefits of modular/class-based decomposition and see first-hand how abstract ADTs are realized with concrete implementations.

When a C++ or Java class is used, this same technique can be used as sketched below for C++.

```
class CreatureImpl;
class Species;

class Creature
{
  public:
      Creature();
      Species * getSpecies();
      void takeTurn();
  private:
      CreatureImpl * myImpl;
};
```

Since we must provide header files, but we want to leave implementation design decisions to the students, we can use this pointer-based implementation, called the *handle-body idiom* in [16]. However, this method has severe drawbacks; for example, students must write a class named Creature, and it is difficult to experiment with alternate definitions of the class.

4.4.3 The Factory Pattern

When we first began to develop the Darwin suite we polled several listserv groups devoted to object-oriented programming in the first years of academic study. We proposed the handle-body idiom as one solution to the abstraction/concrete realization problem here and another solution based on the *Factory* pattern. The Factory pattern is sometimes called a *virtual constructor*. Students create creatures in programs using the following code.

```
Factory * fact = new CreatureFactory;
Creature * c = fact->makeCreature();
```

Student code uses an abstract *Creature* class where abstract now has a technical term: declared specifically as abstract in Java, or having one pure virtual function in C++. The key aspect of an abstract class is that one cannot be constructed, only derived classes of the abstract class are constructed. In the code above, the *CreatureFactory* returns an instance of a class derived from *Creature*. Students must implement a factory class, but the factories are typically instantiated once in main and used thereafter.[2]

The sentiment of those responding to the online query was overwhelmingly in favor of using the handle-body idiom but recognized that the Factory pattern was the better approach. The consensus was that the Factory pattern would be too difficult to explain but that it was clearly the more general approach, and would lead to better designs and understanding in the long run. One of the primary goals of this project is to develop materials that will make the better method recognized as superior in theory and in pedagogical practice.

[2] Factories are often implemented using another pattern, *Singleton*, which makes it impossible to create more than one of each kind of factory in a program.

5 Using Class Libraries

In this section we summarize one method for using patterns and raising the level of design discussion that builds on this framework. A complete description of the libraries, classes, and the method of using these tools is found on our website. As part of this project we have developed a framework of related classes and libraries designed around several patterns. The set of related classes is called *Bargello*.

The class libraries in Bargello include the following.

- A class for parsing command line arguments in C++ and Java
- A pattern and concrete classes realizing the pattern for managing limited resources, e.g., a large number of open files in certain applications.
- A file system abstraction that can be used to traverse directories, tar files, ftp sites, and URLs.

Supporting material on our website shows a technique called "Before and After" that we discovered at the first UP (Using Patterns) conference. Complete programmed solutions to a specific problem are analyzed for strengths and weaknesses with an eye to how the solutions might be improved by using patterns. In our case, student solutions to a problem posed in our Software Design course are dissected (the solutions are anonymous). In many cases the designs are improved by the introduction of appropriate patterns to solve problems that were not solved, or that were solved using methods that in turn lead to other problems. This technique of using patterns makes concrete the idea attributed to both Fred Brooks and Henry Petroski: "*good design comes from experience, and experience comes from bad design.*" The before and after model shows students concretely how designs and programs can be improved by using patterns. The discussion is facilitated by a pattern vocabulary and a higher level of discourse caused, in part, by incorporating patterns into our curriculum.

6 Systematic and Systemic Changes

Current computer science curricula reflect expertise of individual faculty well-versed in subdisciplines of the field. Integration of concepts throughout a curriculum, a so-called spiral approach, is normally accidental rather than planned. This situation will change as an object-oriented approach is adopted. Faculty, especially in smaller schools, will need to become well-versed in what may be a new way of thinking. An analogy applies to recent changes in the curricula of many medical schools. Rather than using a course-based approach: gross anatomy followed by biochemistry followed by clinical perspectives and so on, an integrated systemic approach is followed. A cadaver may serve as the springboard to a large-scale introduction of many related concepts rather than merely as the object of dissection. Problems are put in context, and in the context of whole study rather than isolated subdisciplines.

The same approach is possible in computer science, although this may be unnatural for many faculty. The work that will result from this project will, we hope, make a systematic approach to a systemic curriculum possible at a wide-range of institutions. Several kinds of materials will be developed as part of the project.

- A collection of idioms for C++ and Java that facilitate an object-oriented approach to learning these languages rather than a syntax driven approach. These idioms will be addressed specifically at those new to these languages but who may have experience in other languages.
- A collection of simple programming patterns of the kind outlined in Section 4.1 that focus on general issues not specifically related to object-oriented programming.
- A collection of design patterns that help in designing and implementing larger programs and in learning about object-oriented design and programming.
- Examples that are motivating for both students and educators which provide the context for learning and discussing idioms and patterns. However, examples will be of little help without support for educators in using the examples as the basis for lectures, labs, and assignments. This support, in the form of web-based materials (and perhaps textbooks), is essential for the long-term success of the project.
- Research in how patterns and idioms are best used. This research has repercussions beyond the curricular issues addressed by the previous items. The learning strategies of experienced and novice programmers is a research area that could be addressed by this project as well.

Current trends in computer science education [18] call for the removal of some core courses to be replaced by more modern courses. It is unlikely that faculty will be will persuaded to drop courses for which they assume responsibility, especially when the alternative requires new courses and curricula. A systemic approach to the early courses and the curriculum in general addresses these concerns. The systemic approach, implemented systematically, allows concepts to spiral throughout the curriculum. Revisiting material introduced in one course in later courses makes the material more accessible to students [43]. Although the main thrust of this project is not the systematic reform of the curriculum, the materials developed as part of this project can be used in a systemic approach to the first two years of the curriculum. Such an approach requires the software equivalent of a cadaver. This support must be a framework of projects, explanatory material, and patterns that tie the frameworks together. This proposal addresses these concerns by delivering instructional materials in the form of projects, class and program frameworks, and the materials needed for educators and students to use these materials. The programming frameworks outlined above (e.g., Darwin) provide the

reusable software cadaver we think will help make patterns accessible while providing material for developing interesting problems and assignments.

7 Related Work

There has been some work in using patterns in introductory courses. Wallingford [40, 27] shows how patterns can be used with a procedural and an object-oriented paradigm. This work is part of three-day workshop that will take place in conjunction with ChiliPLoP (see `http://www.agcs.com/patterns/chiliplop/`). These patterns are what we have termed programming patterns, and are typically not used in frameworks or class libraries. Recent work in [19] also addresses patterns in introductory courses. Work at Brown University using Object Pascal in the first course [14] uses patterns implicitly, these patterns are not identified per se. In particular, the patterns are not named which makes them difficult to find and use. This work has been less accessible to the community because of the use of Object Pascal [38]. However, the course has migrated from Pascal to Java which will help make the work more accessible to educators interested in following an object-oriented approach. Several papers address the use of design patterns in data structures courses [21, 26].

8 Summary

Patterns can and should be used to help develop basic and more advanced programming and design skills in academic computer science courses. Simple programming patterns can be used to help acclimate students and educators to the use of patterns, and to build a foundation for continued study of object-oriented design patterns. Object-oriented programming will emerge as the paradigm of choice in academic computer science programming courses. The language used may change, e.g., from Ada to C++ to Java to an as yet unspecified alternative, but the object paradigm is better for many programming endeavors (though not all) than traditional structured programming. To use object-oriented design techniques and languages requires a new way of thinking, one that does not come naturally to many of us who have been mired in a structured programming world. Design patterns help facilitate the transition to an object-oriented way of thinking and, we argue, are essential to use object-oriented techniques correctly and efficiently. Our project is developing materials, available for distribution via the world-wide web, to help educators make the transition to using design patterns. These materials can be used for self-learning, in the classroom, and as part of a more broadly-based spiral approach.

A Web Dissemination

The re-development of material found by searching courses at other institutions shows the promise and the drawbacks of the web as a means of disseminating information, ideas, assignments, and pedagogical modules. Stanford's website [where we first learned of Darwin — see Section 4.4] provided neither code nor libraries for use outside of Stanford. Although the code was made available by contacting the institution, platform incompatibilities can preclude re-using assignments in other contexts. In this case, the graphics library used was part of material developed in conjunction with a book [31], and freely available. However, we found it necessary to develop our own version to support a richer graphics context, to be useful using both Java and C++, and to support an object-oriented approach based on patterns.

Material we make available on the web in conjunction with this project and our courses includes source code, libraries, and instruction. Modules developed as part of this ongoing project include documentation and explanatory material aimed directly at educators. Pattern-based modules require extensive explanatory materials since nearly all pattern books and articles are aimed at those experienced in object technologies.[3]

The materials developed will include guidelines and discussion on how to use the patterns that are shown in concrete programs in new settings. The patterns community recognizes similar shortcomings in the use of patterns as opposed to the discovery and cataloging of patterns [37]. A *Using Patterns Conference* was held for the first time in 1997. The conference was devoted to the use of patterns rather than to their formation and discovery.

Thanks

Robert C. Duvall helped immensely in the development of this paper.

References

[1] Christopher Alexander. *A Pattern Language*. Oxford University Press, 1977.

[2] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[3] O. Astrachan and D. Reed. AAA and CS-1: The applied apprenticeship approach to CS 1. In *The Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*, pages 1–5. ACM Press, March 1995. SIGCSE Bulletin V. 27 N 1.

[4] Owen Astrachan. Pictures as invariants. In *The Papers of the Twenty-Second SIGCSE Technical Symposium on Computer Science Education*, pages 112–118. ACM Press, March 1991. SIGCSE Bulletin V. 23 N 1.

[5] Owen Astrachan. Self reference is an illustrative essential. In *The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education*, pages 238–242. ACM Press, March 1994. SIGCSE Bulletin V. 26 N 1.

[6] Owen Astrachan and Susan Rodger. Animation, visualization, and interaction in cs 1 assignments. In *The Papers of*

[3] Although we hope to develop some of this material into a book, all materials on the web will include complete source, right to use and modify the source, and explanatory text aimed at educators.

the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education. ACM Press, 1998.

[7] Owen Astrachan, Trevor Selby, and Joshua Unger. An object-oriented, apprenticeship approach to data structures using simulation. In *Proceedings of the Twenty-Sixth Frontiers in Education*, pages 130–134, 1996.

[8] Owen Astrachan, James Wilkes, and Robert Smith. Application-based modules using apprentice learning for cs 2. In *The Papers of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*, pages 233–237. ACM Press, February 1997.

[9] Kent Beck. Using a pattern language for programming. In *Workshop on Specification and Design, OOPSLA 87*, 1987. ACM Sigplan Notices 23, 5.

[10] Frederick Brooks. *The Mythical Man Month*. Addison-Wesley, 20th anniversary edition edition, 1995.

[11] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley, 1996.

[12] Marshall Cline. Pros and cons of adopting and applying patterns in the real world. *Communications of the ACM*, 39(10):47–49, October 1996.

[13] Alistair Cockburn. The interaction of social issues and software architecture. *Communications of the ACM*, 39(10):40–46, October 1996.

[14] D. Brookshire Conner, David Niguidula, and Andries van Dam. *Object-Oriented Programming in Pascal: A graphical approach*. Addison Wesley, 1995.

[15] James Coplien. *Software Patterns*. SIGS books, 1996.

[16] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison Wesley, 1992.

[17] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Addison Wesley, 1995.

[18] Alan Tucker et. al. Strategic directions in computer science education. *ACM Computing Surveys*, 28(4), December 1996.

[19] Harriet J. Fell, Viera K. Proulx, and Richard Rasala. Scaling: A design pattern in introductory computer science courses. In *The Papers of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*. ACM Press, 1998.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[21] Natasha Gelfand, Michael T. Goodrich, and Roberto Tamassia. Teaching data structures design patterns. In *The Papers of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*. ACM Press, 1998.

[22] Brandon Goldfedder and Linda Rising. A training experience with patterns. *Communications of the ACM*, 39(10):60–64, October 1996.

[23] G.E. Krasner and S.T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[24] T.D. Meijler and R. Engel. Making design patterns explicit in face: A framework adaptive composition environment. In *EuroPLoP 96*, 1996.

[25] Glenn Meter and Philip Miller. Engaging students and teaching modern concepts: Literate, situated, object-oriented programming. In *The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education*, pages 329–333. ACM Press, March 1994. SIGCSE Bulletin V. 26 N 1.

[26] Dung Nguyen. Design patterns for data structures. In *The Papers of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*. ACM Press, 1998.

[27] Pattern-based programming instruction. NSF DUE-9455736, 1995.

[28] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2:1–9, 1976.

[29] Richard Pattis. Teaching oop in c++ using an artificial life framework. In *The Papers of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education*, pages 39–43. ACM Press, February 1997.

[30] Robert S. Rist. Schema creation in programming. *Cognitive Science*, 13:389–414, 1989.

[31] Eric S. Roberts. *The Art and Science of C*. Addison Wesley, 1995.

[32] Patricia K. Schank, Marcia C. Linn, and Michael J. Clancy. Supporting pascal programming with an on-line template library and case studies. *International Journal of Man-Machine Studies*, 38(6):1031–1048, June 1993.

[33] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[34] Mary Shaw. Patterns for software architecture. In J. Coplien and D. Schmidt, editors, *Pattern Languages for Program Design (PLoP)*, pages 453–461. Addison-Wesley, 1995.

[35] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5), 1984.

[36] Elliot Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9):850–858, 1986.

[37] Jiri Soukup. Implementing patterns. In J. Coplien and D. Schmidt, editors, *Pattern Languages for Program Design (PLoP)*, pages 396–412. Addison-Wesley, 1995.

[38] Andries van Dam. Object oriented programming: Getting it right from the start. NECUSE symposium, 1995.

[39] John Vlissides, James Coplien, and Norman Kerth, editors. *Pattern Languages of Program Design 2*. Addison Wesley, 1996.

[40] Eugene Wallingford. Toward a first course based on object-oriented patterns. In *The Papers of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, pages 27–31. ACM Press, 1996.

[41] Eugene Wallingford. Roundabout: A pattern for writing recursive programs. In *PLoP 1997*. 1997. http://www.cs.uni.edu/∼willingf/research/patterns.

[42] A. Weinand, E. Gamma, and R. Marty. ET++ – an object-oriented application framework in c++. In *OOPSLA '88*, 1988. SIGPLAN Notices, 23(11).

[43] Karl Wender, Franz Schmalhoer, and Heinz-Dieter Bocker, editors. *Cognition and Computer Programming*. Ablex, 1995.