TEACHING DESIGN PATTERNS: A MATTER OF PRINCIPLE

Charles D. Allison
Utah Valley State College
Orem, UT 84058
801-863-6389
allisoch@uvsc.edu

Neil B. Harrison
Utah Valley State College
Orem, UT 84058
801-863-7312
harrisne@uvsc.edu

ABSTRACT

The patterns movement in software design has provided a framework for codifying and communicating solutions for commonly encountered design challenges. Design patterns are not intended to be only entries in a cookbook, however; they emerge by following good design principles to balance the forces present in a software development context. According to the authors' experience, design patterns are best presented in concert with the design principles that bring them to life. This paper discusses the interrelation between design patterns and principles and reports on a pilot course in teaching the principles and patterns of software design that, after two successful semesters, has recently been adopted as a requirement in a newly inaugurated bachelor's degree in software engineering.

INTRODUCTION

One of the significant challenges in CS education is teaching not only how to write software, but how to write *good* software. It doesn't come naturally: left to their own devices, students often write convoluted, complex code that is a nightmare to debug and impossible to maintain. We can teach principles of good design, but this is difficult to get across effectively. The principles can be difficult to learn and remember, particularly if they are not linked to motivation for their use. Learning design patterns can help students learn and remember the basic design principles. We begin with the example that introduces the course: implementing a stack of fixed size.

CS students learn early to separate interface from implementation. The first evidence appears when they make data and other implementation details *private*, so that users of the classes they create don't explicitly depend on internals. They eventually learn, however, that an implicit dependency still remains: any change to implementation detail may still cause a need for rebuilding an application. This can be seen in a from-scratch, fixed-size stack implementation. A Java sample follows.

```
public class FixedStack {
    private Object[] data;
```

```
    private final int max;
    private int count = 0;
    public FixedStack(int max) {
        this.max = max;
        data = new Object[max];
    }
    public void push(Object x) {
        if (count < max)
            data[count++] = x;
        else
            throw new BufferOverflowException();
    }
    public Object pop() {
        if (count == 0)
            throw new BufferUnderflowException();
        return data[--count];
    }
    public Object top() {
        if (count == 0)
            throw new BufferUnderflowException();
        return data[count-1];
    }
    public int size() {
        return count;
    }
```

Figure 1 – A FixedStack Implementation

The dependency a user has on `FixedStack` may not seem too onerous—if the code for the implementation of `FixedStack` changes, then its new *.class* file just needs to be installed and the application restarted. But what if a better fixed-stack class with a different name comes along? Client programmers will have to do a global search-and-replace and then recompile.

A better solution is to *truly* separate the implementation from the interface, by encapsulating the interface in a separate abstraction, leaving clients to program to the interface only. A suitable interface can be defined as follows:

```
public interface IFixedStack {
    void push(Object o);
    Object pop();
    Object top();
    int size();
}
```

Users can now employ any type that implements the `IFixedStack` interface without changing their code.

DESIGN PRINCIPLES

Experienced designers follow rules of thumb learned by their own experience and that of others to create quality software. Many such principles have been collected in well-known works. [2,3,4,5] The underlying design principle followed in the `FixedStack` example above is:

Program to an interface, not an implementation.[1]

Adhering to this principle shields users from changes in implementation.

Now suppose there is a need to replace fixed-stack implementations at *runtime*. The following `IFixedStack` wrapper class suffices.

```
public class FixedStackStrategy implements IFixedStack {
    private IFixedStack impl;
    public FixedStackStrategy(IFixedStack fs) {
        impl = fs;
    }
    public void push(Object o) {
        impl.push(o);
    }
    public Object pop() {
        return impl.pop();
    }
    public Object top() {
        return impl.top();
    }
    public int size() {
        return impl.size();
    }
    public void setImpl(IFixedStack fs) {
        impl = fs;
    }
}
```

This class illustrates the *Strategy Design Pattern* which conforms to the following important design principle:

Encapsulate the things that vary, separating them from the things that stay the same. [1]

Interfaces tend to "stay the same" (more so than implementations do, anyway), so the first principle mentioned earlier is actually a special case of this more general principle.

DESIGN PATTERNS

Design patterns are conceptual solutions to recurring design problems that adhere to principles of good design that have emerged over the decades. A design pattern contains at least the following information:

Summary	Describes the pattern in succinct, high-level terms.
Problem	Describes the problem, including the forces that cause the "dilemma", and the negative consequences of not changing the design.
Solution	Describes a solution to the problem that resolves the forces in a positive way. Often includes a sketch.

Figure 2 – Template for a Minimal Description of Design Patterns

The best known, but far from the only design patterns are documented in [1]. Documentation for Strategy might include such information as:

Summary	Define a family of algorithms, encapsulate them, and make them interchangeable. Strategy lets algorithms vary independently from clients that use them.
Problem	A class may need variants of an algorithm/implementation configurable at runtime. Or, there may be many related classes that differ only in their behavior. This can lead to significant amounts of code that must change when an algorithm or implementation changes.
Solution	Define an interface for the algorithm family. Encapsulate each algorithm/implementation as an implementation of its interface. The abstraction that uses the algorithms keeps polymorphic references to their implementations.

Figure 3 – Minimal Description of the Strategy Design Pattern

A design pattern can be seen as a realization of one or more design principles in an object-oriented context. Strategy realizes the two principles mentioned earlier by reifying the relationship between the usage context and its associated implementation hierarchy. Likewise, the Decorator pattern realizes the following design principle:

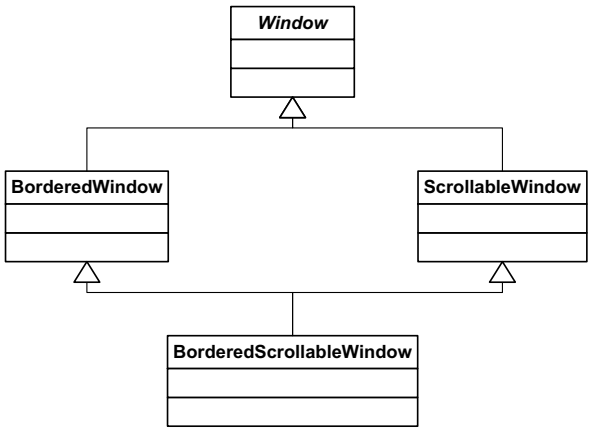
The Open-Closed Principle: *A class should be open for extension but closed for modification.*[5]

This principle implies that one should be able to add to or replace the functionality of a base class without changing its code. A decorator type implements the same interface as the class it decorates, and a decorator object has a reference to the specific object it decorates. This allows decoration to occur at runtime and multiple decorators to be nested. The Java I/O package is a well-known example. Basic stream objects can be "decorated" with additional functionality, such as buffering or formatting. A declaration such as the following is idiomatic in Java:

```
BufferedReader br = new BufferedReader(new FileReader("file.dat"));
```

The `BufferedReader` object wraps an existing `FileReader` to provide buffered input and also adds a `readLine()` method. Both classes implement the `java.io.Reader` interface.

The Decorator design pattern shows the advantage composition has over inheritance. Adding capability through inheritance can lead to a combinatorial explosion of class definitions. For example, suppose you have a class, `Window`, and need to add borders and scrolling. A novice might do something like the following. [6]



A little algebra reveals that for every new window attribute to be added, the number of classes grows exponentially (you need 2^n classes altogether for *n* attributes). By using a decorator class for each attribute, the number of classes is linear, and the resulting design is more flexible, since decorators can be composed as needed at runtime. Thus Decorator illustrates another important object-oriented design principle:

Favor composition over inheritance.[1]

STRUCTURE OF THE COURSE

The Computer Science department at Utah Valley State College has offered a junior-level course entitled, "Principles and Patterns of Software Design", since Fall 2005. The prerequisites are CS21 and a junior-level programming course in Java, C++, or C#. The course uses "Head First Design Patterns" [6] as a text because of its emphasis on the relationship between patterns and principles, and because it introduces them in a natural, incremental fashion. Both C++ and Java are used in examples, since the former has useful static realizations of design patterns by virtue of its template facility. The course also introduces the following patterns not found in the main chapters of the text: External Polymorphism and Visitor. The following principle not found in the text is also examined:

The Interface Segregation Principle: *Clients should not be forced to depend on methods they do not use.* [7]

In other words, *interfaces belong to clients*, not implementations.

Students are allowed to use any statically-typed object-oriented programming language for projects. Most students use C++, Java, C#, or D.

Typical programming assignments have included:

- Write a program that monitors a (simulated) live stock market feed, producing various summary reports on demand. (Observer)

- Write a “filter iterator” that takes an arbitrary predicate and only yields elements of a sequence that satisfy the predicate during traversal. (Iterator)
- Keep a log of database commands and run them in batch mode. Support an “undo command”. (Command)
- Simulate directory management commands as found in typical hierarchical file systems (mkdir, cd, rm, rmdir, etc.). (Composite, Iterator, Visitor)
- Add caching and security to a simple database API. (Proxy)

This course has been a significant success, so much so, that it has been made a required course in the curriculum for our BS in software engineering. Students have reported that they use the principles and patterns in large projects in subsequent courses, such as compiler construction and advanced architecture. An important side benefit is that, having studied a number of patterns (about seventeen so far), they have acquired a pattern vocabulary that enables effective discussion of design issues at a conceptual level (e.g., “this is a good place for an adapter”, etc.). This vocabulary constitutes a “design language” applicable in projects that involve object-oriented techniques.

SUMMARY

It is challenging to teach design principles effectively. Design Patterns are an effective vehicle for teaching good design principles. Instead of merely memorizing the principles, students see them in action and thus are more motivated to search out their meaning and benefit. Our students have remarked that this course has “opened their eyes” to what software design is all about, and they now see how to apply sound design principles in all their software efforts.

1. Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
2. Yourdon, E. and Constantine, L., *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Prentice-Hall, 1979.
3. McConnell, S., *Code Complete*, Second Edition, Microsoft Press, 2004.
4. Riel, A., *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.
5. Meyer, B., *Object Oriented Software Construction*, Prentice Hall, 1988, p 23.
6. Freeman, E. and Freeman, E., *Head First Design Patterns*, O’Reilly, 2004.
7. Martin, R., *Agile Software Development: Principles, Patterns, and Practices*, Prentice-Hall, 2002.