Much Ado About Patterns

by Robert Zubek

Introduction

It is no news that patterns abound in all areas of human endeavor. In programming in particular, one can often find certain regularities in the way things are done -- there are canonical and time-tested ways of designing compilers, operating systems, and many other types of complex software. On a lower level, there are also many regular ways of dealing with recurring problems, such as that of how to represent complicated data structures in memory. Indeed, it seems that everywhere we look in computer programming, we are surrounded by regularities and standards.

Considering all this, it might seem that the recent excitement over patterns is just ``much ado about nothing'' -- after all, haven't we been using patterns in software development all along?

After taking a closer look, we notice that that is not the case. What causes a stir in the software development community is not just regular patterns in code, in the sense of simple regularity and repeatability. Rather it is the more abstract concept of a pattern as a *formalized way of recording experience*. This new way of thinking about patterns was introduced by Christopher Alexander, an architect who used patterns in architecture and urban planning -- he introduced the idea in [1], his 1964 book, and developed it further in later works such as [2]. The immense potential of patterns has recently been recognized in the software community as well, mainly because they can be used to simplify the creation of complex systems.

If the concept of a formalized pattern seems a bit puzzling, it probably should, because patterns -- in this new sense -- can be powerful beyond expectations. We will, for example, find that such patterns not only can provide us with solutions to specific problems (which is what recognizing regularities in code can help us to do), but can also teach us how to deal with similar problems in the future, or even help encode information about the design of the program in its implementation -- embed explanation of code into code itself!

In this article, I would like to present and explain patterns, with special emphasis on design patterns, a type of pattern often used in software design. We will begin with a brief overview of what patterns are -- just enough to get us going, because a more detailed definition might be a bit too abstract at first. Then we will see a few examples of patterns -- what they are, how they are used, and when they can be useful. After that we will return to the problem of defining a pattern, and examine the more detailed definitions of patterns, as well as some of their common characteristics.

What Are Patterns? (Part I)

The fundamental question ``what are patterns?'' has no absolute answer -- there is no *exact* definition. Were it otherwise, Alexander would have been able to describe them in a sentence or a paragraph, rather than the several books that he has written so far. There are some generalizations that can be made, however, and the people who

work on patterns have more or less agreed on a certain way of defining them. We will delay a full definition until we see some examples; for our immediate purposes, we can use a popular simplified definition, which says that a **pattern** is a *solution to a problem in a recurring context.* This means that a pattern is a bit of formalized experience in dealing with particular recurring situations. However, all three elements of the definition are necessary: the solution has to be provided, the problem must exist, and the context must not be unique. This definition is not to be understood from the first reading -- it's just something to keep in mind while reading examples.

What Are Design Patterns?

Much of what is being said about patterns in software development pertains to **design patterns**, which are a group of patterns that apply to the task of creating software systems. The label ``design patterns'' is somewhat misleading, considering that it can be used to describe any patterns involved in designing anything (not just software). However, the label was used in some highly influential works (such as [4]), and it remained. [4] is also the source of a very useful definition of design patterns: ``descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.''

Sample Design Patterns

The following examples represent design patterns in object-oriented programming. As you read through them, notice certain regularities between them. These patterns have been introduced in [4], from which I took their definitions. Note that due to the introductory nature of this article, as well as space constraints, these patterns have been shortened and simplified. After all, it took Gamma *et al* 268 pages to explain 23 patterns!

These examples are in C++ (or pseudo-C++), and they require some knowledge of the language and object-oriented design in general. They should not be hard to replicate in other object-oriented languages (though one must remember that while patterns are language independent, not *all* languages are powerful enough to make using patterns worthwhile [5]).

While reading, you might find it useful to jump between the sample code and its description. Figure 1 will explain the notation used.

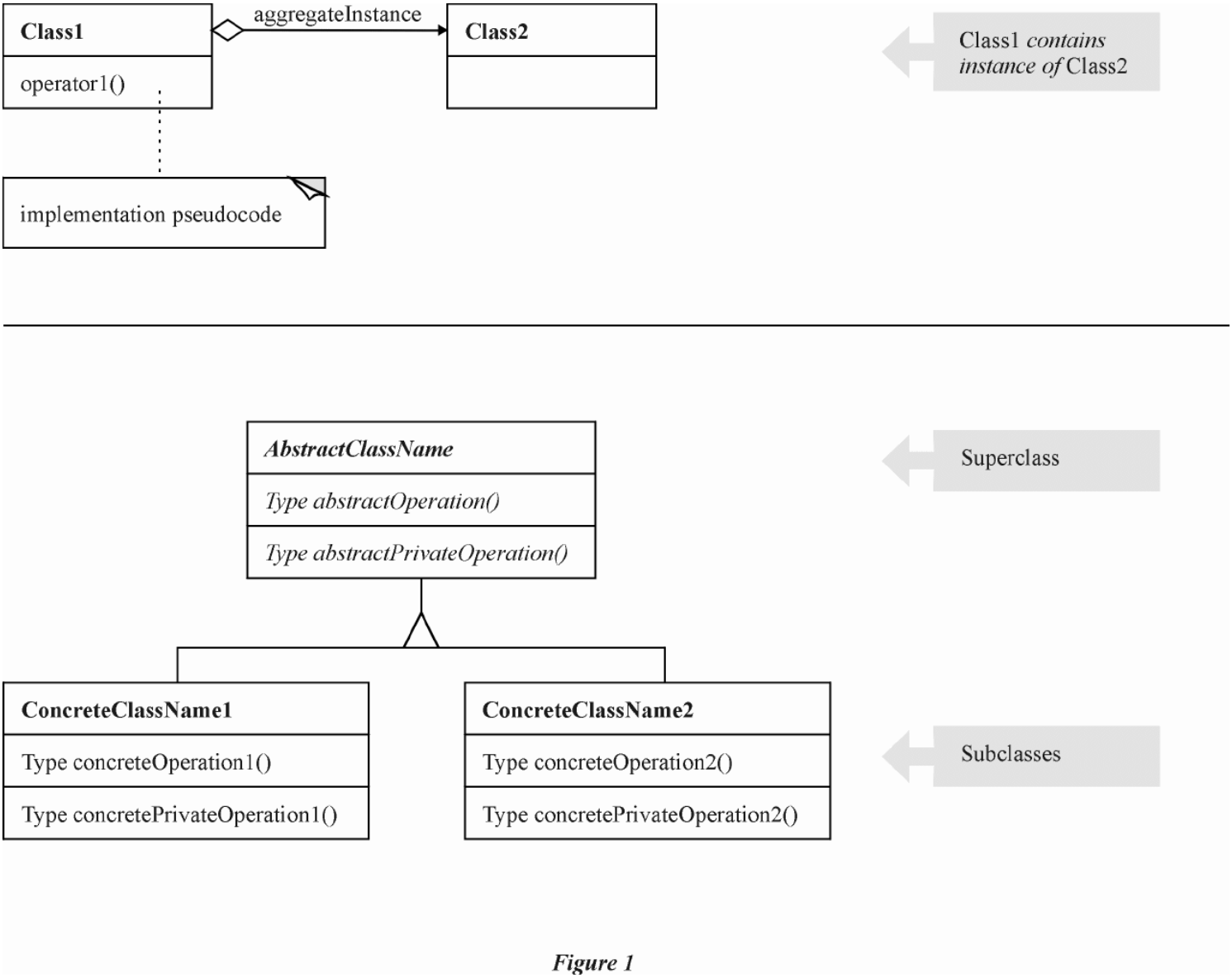

Figure 1

Example 1: Strategy [4]

Intent:

> The Strategy pattern encapsulates algorithms into interchangeable objects. Those objects can vary independently from the clients that use them.

Motivation:

> It often becomes necessary to use several interchangeable algorithms in a program. An example of this might be a data compression utility that must support different algorithms with different time and space tradeoffs. Another case might be an image editor, which may need several algorithms for handling different image formats.
>
> The easiest solution is to hard-code all algorithms into the program, using a command such as `switch`in C to pick one at runtime. This, however, is highly undesirable -- such code is hard to maintain and all the

algorithms will be required even if the user never makes any use of them. An elegant solution to these problems is to encapsulate those algorithms into different (but interchangeable) classes that can be instantiated when needed. The name we will use for such an encapsulated algorithm is **strategy**.

Structure:

Figure 2 presents the objects that are involved in this pattern, and the relationships among them.

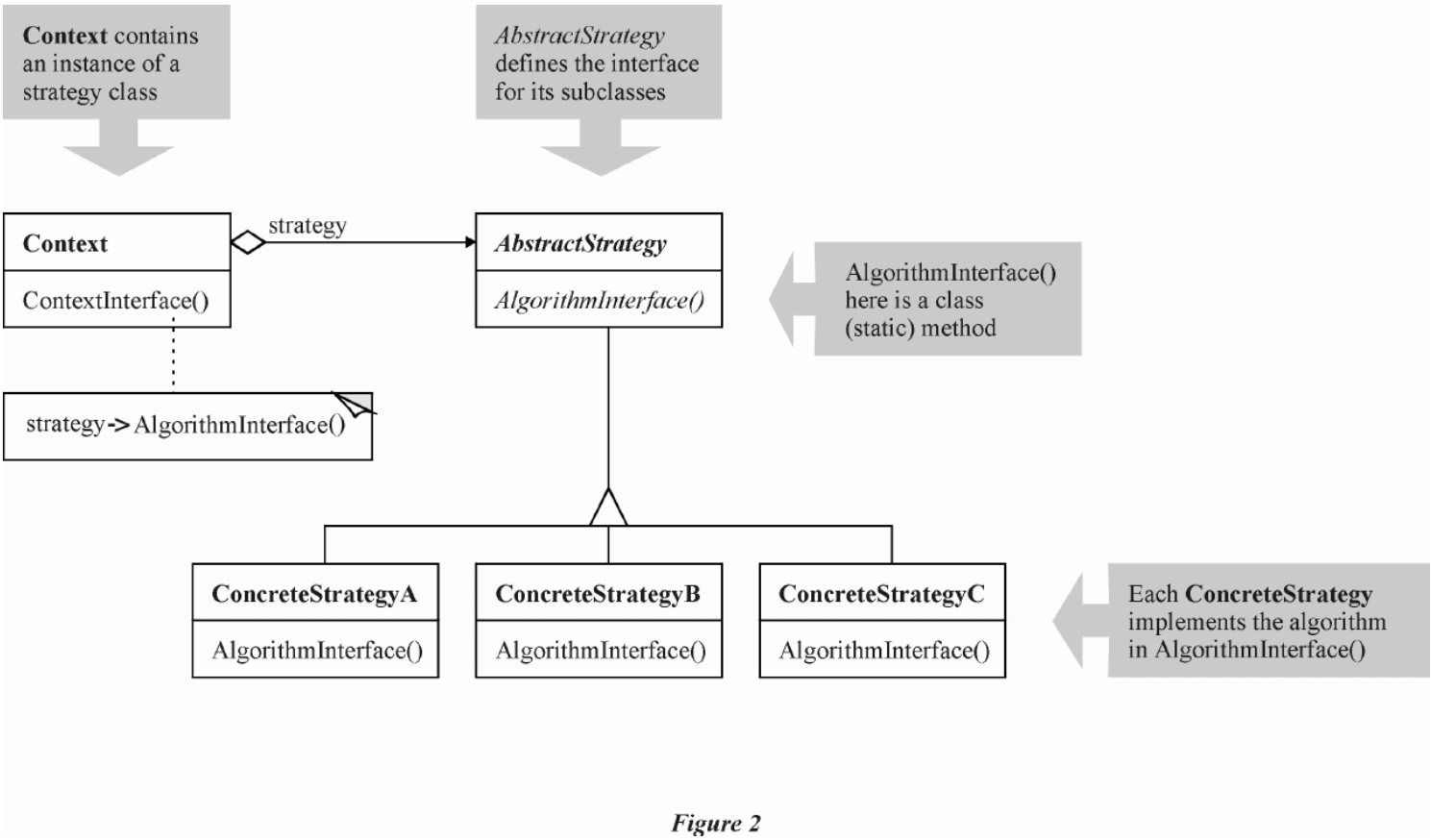

Figure 2

Figure 2

In this diagram, `AbstractStrategy` is an abstract class that defines a common interface for all strategies. Part of that interface is method `AlgorithmInterface()`, which has to be implemented by all subclasses. Each `ConcreteStrategy` is a subclass of `AbstractStrategy`, and implements the interface methods -- a `ConcreteStrategy` is thus the actual encapsulation of an algorithm.

`Context()` is the procedure that makes use of the multiple algorithms, and it contains an instance of one of the strategies. When the algorithm needs to be applied, `Context()` calls the `ContextInterface()` method, which calls the `AlgorithmInterface()` method of the instance of the strategy class.

Example:

Suppose we want to implement several interchangeable sorting algorithms. We choose the class

`Sorter`to represent the main program. We also have several classes representing different sorting algorithms: `SortingStrategy`is the abstract class that defines the interface, and `BubbleSortStrategy`, `HeapSortStrategy`, and `QuickSortStrategy`are its subclasses. `SortingStrategy`defines an abstract method, called `doIt()`, which is expected to do the sorting, and the subclasses are expected to implement this method using the appropriate algorithms. Notice that the `List`class, which is an argument to `doIt()`, is not defined here; it is just some data structure that represents the list of objects to be sorted.

We let `Sorter` contain an instance of one of the strategies. `Sorter` will also have a method called `sort()`. Whenever the list needs to be processed, the user calls the `sort()` method of `Sorter` -- that method looks up the strategy that `Sorter` specifies, and passes the list of the objects to the `doIt()` method of that strategy, which sorts it.

Here is how a simple implementation of sorting algorithms might look:

```
class SortingStrategy {
public:
  virtual void doIt (List *someListOfElements) = 0;
};

class BubbleSortStrategy : public SortingStrategy {
public:
  void doIt (List *someListOfElements) {
    // ... list gets sorted here using bubble sort ...
  }
};

class HeapSortStrategy : public SortingStrategy {
public:
  void doIt (List *someListOfElements) {
    // ... list gets sorted here using heap sort ...
  }
};

class QuickSortStrategy : public SortingStrategy {
public:
  void doIt (List *someListOfElements) {
    // ... list gets sorted here using quicksort ...
  }
};
```

We must not forget about `Sorter`, of course.

```
class Sorter {
public:
```

```
  void sort(List *listOfItems) {
    if (strategy != 0)
      strategy->doIt (listOfItems);
  }
  SortingStrategy *strategy;   // pointer to the instance
}
```

For those not familiar with the details of C++ syntax: declaring a method as ``virtual ..... = 0;" means that it's an abstract method, and defining it is the responsibility of the subclasses.

How would this code work? First the programmer must explicitly select which algorithm will be used by instantiating the proper subclass:

```
Sorter->strategy = new QuickSort;
```

Then we just call the `sort()` method whenever data needs to be sorted:

```
Sorter->sort(someList);
```

Notice that whenever we need to change the algorithm, it's as easy as instantiating another strategy class. Even after we change the algorithm to be something else, we still use the same `sort()` method to start sorting. This way we make the process of sorting independent of the actual algorithm.

Consequences:

It turns out that there are several benefits to using this pattern:

- The Strategy pattern lets programmers group similar algorithms into one family. This should lead to easier recognition of common functionality.
- The pattern provides an alternative to subclassing: one could just as well subclass `Sorter` into `BubbleSorter`, `QuickSorter`, etc., but this approach fails when we try to do the same for several different families of algorithms simultaneously. For example, when we need separate families for sorting, searching, and displaying, we would need a separate subclass for each of the combinations of algorithms.
- The pattern is also an alternative to conditional statements: it eliminates the need for big statements such as `switch` in C, which are generally regarded as bad style.
- It is also a big improvement over putting algorithms in different procedures because separate points of entry for different algorithms are not needed; when using the pattern, only one point of entry, `doIt()`, needs to be known.
- The pattern provides a choice of implementations: several different algorithms that accomplish the same goal can be encapsulated in this manner, letting the user pick the one most favorable at the time (for example, letting the user pick between one that is fast but space-consuming, or slower but space-efficient).

There are also several drawbacks, however:

- If it is not the user who picks the strategy, the client will have to be aware of differences in strategies in order to select the best one. This might muddle modularity, as clients might need to know something about the implementation of the strategies.
- Because some strategies might not need any or all information required by `AlgorithmInterface()`, there might be some unnecessary overhead in communication between strategies and the client.
- This pattern will increase the number of objects in the system, which in some cases might be undesirable.

Example 2: Singleton [4]

Intent:

To create a class that can have *only one* instance, available globally in the code.

Motivation:

It is sometimes useful to create a class that can only have one instance. A file system is a good example -- there should be only one filesystem. Normally, a program could create as many instances of a class as it requests.

It is also important to be able to easily ``locate" that instance. Creating a global pointer is undesirable for several reasons:

- Namespace pollution: unique names for both the class and the global variable are required.
- Global objects are created at initialization time, we might want to wait to create the singleton until some later time (we might not have all the required data available at first, for example).
- C++ does not guarantee the order in which global variables are initialized; therefore one cannot create dependencies between different global variables [8].

Structure:

Figure 3 illustrates the structure of this pattern.

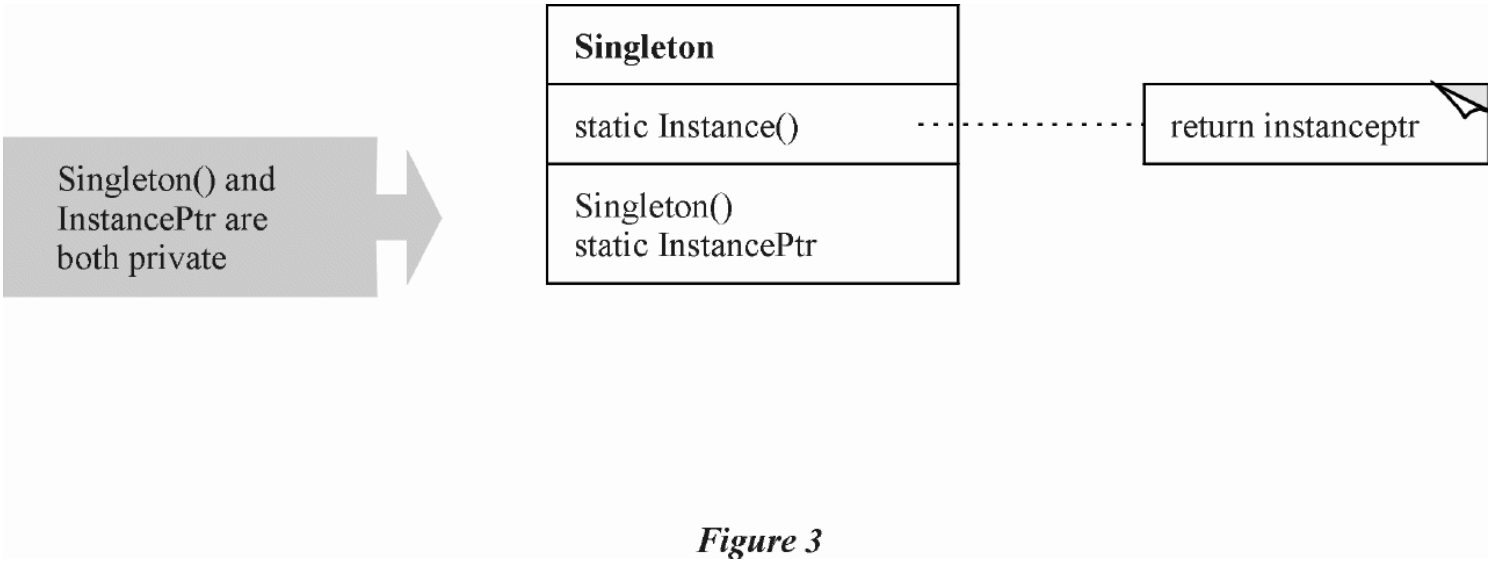

Figure 3

Figure 3

The constructor is not accessible from outside of the class, so the user cannot directly create an instance of `Singleton`: creation of an instance must be done from *within* a method in that class.

The method that will take care of instantiating the singleton is `Instance()`. It will also make sure that not more than one instance ever gets created. Once an instance exists, `Instance()` returns a pointer to it.

In this design, `InstancePtr` is a static variable, accessible only within the class, and it's used to store the pointer to the instance of the singleton.

Example:

Let us take a look at how we might implement a keyboard buffer class, to store all the keystrokes that were typed in by the user and are waiting to be processed by a program. We simplify this example by assuming that putting keystrokes into the buffer happens automatically, and that the only function the buffer supports is `getch()`, which returns the oldest byte (or keystroke) from the buffer.

We can ensure that the instance of the singleton is unique by intercepting requests to initialize an instance of the class. We do that by making the constructor a *private* method -- that is, one that can only be accessed by other methods in that class. Since all initializations are done through the constructor, by making the constructor unavailable to the user we make sure that the user will never directly create a second instance.

Now we have a problem, namely, only other methods within that class can access the class constructor. However, those methods would themselves have to be part of some instance of the class. This would seem to indicate that we would have to have an instance of the class to create the first instance, which is a contradiction.

Fortunately, that is not correct. Object-oriented programming provides us with methods and variables that are part of the class, but that can be accessed without any particular instance of the class. They are often called **class methods and variables**, although when dealing with C++ the term ``static methods and variables'' is also commonly used. A class variable is a ``part of a class, yet is not a part of an object of that class,'' which means that a class variable can be accessed outside of any instance [8]. In many respects it's like a global variable, but it's a part of the class, and thus its availability to code outside the class can be easily limited (by making it private, for example) and it can access the private and protected members of the class. Similarly, there are class methods, which do not depend on whether an instance of the class exists, and behave like global procedures.

Thus, declaring `Instance()` to be a class method easily solves our problem. This way, `Instance()` can be called at any time, and by giving it access to the constructor we give it the ability to instantiate the singleton class. We must make sure, however, that it does not make multiple instances of the class, so we keep a pointer to the instance; if the pointer is null, `Instance()` creates an instance of the class, but if an instance already exists, it just returns the value of the pointer.

This suggests the following definition for the class:

```
class KbdBuffer {
public:
  char getch();  // reads from the buffer
  static KbdBuffer *Instance();  // returns pointer to instance
private:
  KbdBuffer();  // the constructor
  static KbdBuffer *instancePtr;  // pointer to instance
};

KbdBuffer *KBdBuffer::instancePtr = 0;  // reset pointer

KbdBuffer *KbdBuffer::Instance () {
  if (instancePtr == 0)  // if pointer is null...
    instancePtr = new KbdBuffer; // ... create instance
  return(instancePtr)
}
```

We can see how this would be used in a sample program: if the programmer wants to query the keyboard buffer, he or she need only write:

```
char foo = KbdBuffer::Instance() -> getch();
```

Notice that this will create the instance if necessary before returning its pointer, and only one instance will ever exist.

Consequences:

- Because only one instance of this class can exist, it gives the programmer control over how the clients access it.
- The namespace is cleaner because only one name (the class name) is required (instead of one for the class plus one for each global variable).

  There are also several benefits to an extended version of the Singleton pattern, which is detailed in [4].

What Are Patterns? (Part II)

Now that we've seen some examples of patterns, let's take a look back at our working definition. Thinking of a pattern as a *solution to a problem in a recurring context* now might seem a bit problematic. First of all, the emphasis is on a *solution*, but a pattern is more than that -- it does not just provide a cookbook answer, a trick, or an algorithm, but instead it teaches how to deal with similar (and not-so-similar) situations in the future. One of the lessons the Singleton pattern teaches, for example, is that whenever we don't want the user to create an instance of a class directly, we can hide the constructor behind the private statement, and provide some class method for instantiation.

Second, *problem* is defined too narrowly. In the Strategy pattern, for example, there was no single problem to make us use the pattern instead of hard-coding the algorithms. Rather, there was a set of constraints, or *forces*, that made the pattern more beneficial -- forces such as the ease of maintenance, reduction of memory usage, or extensibility. Thus, we must understand that the problem is not a single entity, but rather a set of forces that often include the need for correctness, optimized resource usage, structure, extensibility, and so on [7].

Third, the *context* must refer to a recurring set of situations in which the forces apply. The pattern would have no value if the situation in which it is applicable occurred only once in a thousand years -- the time span has to be short enough to allow people not only to notice a good pattern, but also to benefit from applying it during the next occurrence of the context. Similarly, it has to be the context in which the same set of forces applies -- a pattern is no good if the context recurs but the forces keep changing.

It is advantageous to find unique names for patterns. These names will serve to distinguish between individual patterns and can also serve to form a dictionary of concepts that developers might use when designing systems and communicating [9]. The names therefore form a ``pattern language'' that enhances the developer's ability to reason about complex problems.

Thus, we can use the above definition only if we appropriately expand the terms ``solution'', ``problem'', and ``context''. Unfortunately, there are no definitions that would compress these qualifiers into a short sentence. However, a longer and comprehensive definition would do quite well, such as the following one suggested by Appleton in [3]:

> *A pattern is a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and a system of forces.*

While the pattern community has not agreed on one precise definition, as long as we keep all these qualifiers in mind, we will have a good idea of what patterns are.

How Can I Benefit From Patterns?

There are several ways in which using patterns is beneficial to all types of programmers. The most obvious benefit is that patterns provide solution to problems, so they save development time. Patterns are also ways of recording experience, and thus learning patterns improves (or solidifies) one's skills in that area.

A less obvious characteristic of patterns, which -- as many developers stress -- is nevertheless very important, is that a set of patterns creates a shared common vocabulary among developers. Using this ``pattern language'', developers are able to share experience and communicate effectively about high-level concepts, which is very beneficial considering the complexity of large software projects. By presenting a common set of metaphors, patterns work also in creating a common way of thinking about problems and solutions in software development.

As for design patterns in particular, they can make code easier to read. By recognizing a pattern in code, one sees not only a solution to a problem, but also the set of forces and the context in which they exist. Thus, when the reader of the code notices a design pattern, he or she also gets some information about the relationships between the parts of the system that led to this pattern being beneficial; indeed, it seems that some description of the design has been transferred into the code by using the pattern.

Conclusion

Patterns in general, and design patterns in particular, have become a very discussed topic in the fields related to software development. They represent many of the things we, as software developers, are looking for: easier (and more formalized) ways of sharing experience, simplifications of complex systems, and ways of writing code so that it also conveys information about the overall design. Their popularity, however sudden, is well deserved.

Patterns represent a big shift in software design, which should make programming not merely more efficient, but also more clear.

Pointers

It is my hope that after reading this article you became at least interested in (if not completely fascinated by) patterns. If you would like to learn more about design patterns, as used in software development, a very highly regarded book is [4], and titles of more books can be found in [6]. Many links to online documents about patterns in general, such as [3], [7] or [9], can also be found on the Pattern Homepage (hillside.net/patterns/patterns.html)

Acknowledgements

Special thanks go to Bryan Quinn for his extensive comments on the draft of this article.

References

1

Alexander, C. *Notes on the Synthesis of Form.* Harvard University Press, Cambridge, Mass., 1964.

2

Alexander, C., Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, New York, New York, 1977.

3

Appleton, B. *Patterns and Software: Essential Concepts and Terminology.* http://www.enteract.com/~bradapp/docs/patterns-intro.html

4

Gamma, E., Helm, R., Johnson R., and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass., 1995.

5

Grosso, W. Dynamic Design Patterns in Objective-C. *Dr. Dobb's Journal*, 268 (Aug. 1997), p. 38.

6

Kaplan, J. Patterns Reading List. *Object Magazine* (Mar. 1997), http://www.sigs.com/publications/docs/objm/9703/9703.patterns.html

7

Lea, D. *Patterns-Discussion FAQ*. http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html

8

Stroustrup, B. *The C++ Programming Language, 3rd Ed*. Addison-Wesley, Reading, Mass., 1997.

9

Vlissides, J. Patterns: The Top Ten Misconceptions. *Object Magazine* (Mar. 1997), http://www.sigs.com/publications/docs/objm/9703/9703.vlissides.html

Robert Zubek is a junior at Northwestern University, majoring in computer science, and a co-founder and past president of the local ACM student chapter. His current activities include advocating Perl, digging through Knuth's trilogy, and dreaming about all the software he could have written if he didn't have to sleep -- but above all, he enjoys spending time with friends and family.