Language Features Meet Design Patterns: Raising the Abstraction Bar

Judith Bishop

Computer Science Department
University of Pretoria
Pretoria 0002

jbishop@cs.up.ac.za

ABSTRACT

In the context of software engineering, abstraction is the means by which we move from layer to layer in the realization of the solution to a large problem. It has been recognized for over a decade that design patterns are one of the key mechanisms for implementing reliable and maintainable software. This paper explores where they fit in in the software "food chain". In particular, it examines how advances in language design can narrow the gap for implementing design patterns. Examples are given of syntax features in C# 3.0 (extension methods and LINQ) as well as of library methods (Serialize) in terms of which pattern implementations become easier to produce and reproduce. The challenges that face design pattern implementation are discussed and the promise of reusable design patterns examined.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – Classes and objects, Inheritance, Patterns

General Terms

Algorithms, Design, Reliability, Human Factors, Languages.

Keywords

Design patterns, language features, C# 3.0, LINQ

1. INTRODUCTION

Abstraction is defined as a domain-independent unit of a design vocabulary that subsumes more detailed information [15]. For thirty years, this principle of abstraction has fed into the improvement of computer software. Major milestones are structured programming, data abstraction (later becoming object-orientation), and generic (or template) programming. Each of these movements led to software that was more reliable and more reusable. Beneficial side effects were that programming also became easier to describe as a discipline, and to teach. For the more complex discipline of software engineering, however, the incorporation of abstract thinking has not been as obvious or as codified.

ROA'08, May 11, 2008, Leipzig, Germany.

Kramer [13] regards abstraction as the key to computing and believes that it is necessary to measure the abstraction abilities of those entering the profession. He further emphasizes that abstraction occurs at different levels and that it is important to be able to move between levels [12]. This paper explores this point by looking at the relationship between design patterns and programming languages. A decade ago, there was considerable discussion about raising the bar so as to provide direct support for design patterns in language features [2] [5] [6] [7] [10]. However, the languages of the time (chiefly C++) did not provide sufficient abstract features to enable this movement to gain momentum. With the advent of C# 3.0, the situation has changed and there is now definite evidence of how the power of the language can be used to implement design patterns in new and higher-level ways [4].

The issue then becomes: how do these implementations become part of the design vocabulary talked about by Rugaber [2006]? If one scans books on design patterns published over the past decade, one finds that the implementation of the patterns has not changed very much. Whereas languages have introduced higher-level features such as generics, delegates and iterators, pattern implementations still rely mostly on inheritance and aggregation for linking the classes that play the roles in a pattern. In other words, abstraction is not being actively used in the field of design patterns. Part of the reason for this position is that design patterns are themselves a "vocabulary that subsumes detailed information". The common vocabulary of patterns is one of its most valuable assets. Yet, it is in the nature of patterns as *design* elements that they have to be *implemented* over and over again. Meyer has done extensive research into what would make a pattern componentizable, so that they can be reused [1] [14]. More recently, aspects have been seen as a way of maintaining the higher layer of abstraction of patterns [11]. However, the vast majority of developers and software engineers sees patterns from the level of UML diagrams. They then realize them in whatever implementation language is dictated by custom or availability.

This paper seeks to show how by making use of the more abstract features of a programming language, the gap between design patterns and their implementation can be narrowed. The result is faster and more secure implementation, precisely those welcome attributes that the abstraction milestones mentioned earlier achieved. Although I shall present examples in C# 3.0, the thrust of the argument is how to educate and encourage developers to move away from established practice in any language, present or future. Thus I shall present an overview of my experience with patterns, and then concentrate on three – Bridge, Prototype and Iterator. In each of these three cases (but not only these), the C# 3.0 implementation is considerably shorter than the equivalent

standard C# or Java version. I shall conclude that the use of the available language abstractions therefore contributes to readability, writability, extensibility and traceability. Full versions of all the patterns can be found in at http://patterns.cs.up.ac.za.

2. THE SOFTWARE FOOD CHAIN

2.1 Background

Pioneers in the field of ecology in the early half of the last century recognized that the animal kingdom existed in layers in terms of the food they eat, the now familiar food chain [8][1]. Fascinating results from that study are:

1. With each shift from one link to the other, up the chain, body sizes are larger, and population sizes are smaller.
2. Each stage in the chain transforms smaller particles into larger units, thereby making the food conveniently available to still larger animals, who couldn't cope with the smaller particles.
3. There are very definite limits, both upper and lower, to the size of food that any animal can eat.

These properties are the same as those that we recognize, or hope to achieve, in abstraction in software. As software is implemented in layers of abstraction, the upper layers contain larger "bodies" or components, and fewer of them. The process of implementation creates new items that certainly do form a more convenient set of "particles" or components for the next layer to use. Finally, abstractions need to be carefully chosen in order to be useful. At any one layer in software development, if the lower layer is too low, the task of the developer becomes unnaturally difficult. Similarly, if the elements provided are too highly abstract, they could well be ignored in favour of those in the middle ground.

We now apply these principles to the relationship between design patterns and programming languages.

2.2 Design patterns as abstractions

Design patterns were introduced in 1994 in a book that specifies and describes 23 patterns [9]. These form the foundation of any study of the subject and are still regarded as the essential core patterns today.

> *Design patterns encapsulate common ways of using language features together.*

The core patterns address issues in mainline object-oriented programming (OOP), and the original implementations were presented in C++ and Smalltalk (the primary OOP languages at the time they were developed). Since then, other books have implemented the patterns in Java, Visual Basic, and C#. As the value of the pattern concept has become accepted, new patterns have been proposed to add to the original list. In addition, there are now patterns that are applicable to specific areas, such as software architecture, user interfaces, concurrency, and security. Although these patterns are extremely important in their areas, their adherents are fragmented, and the core set of universally accepted patterns has not been expanded.

Design patterns provide a high-level language of discourse for programmers to describe their systems and to discuss common problems and solutions. This language comprises the names of recognizable patterns and their elements. The proper and intelligent use of patterns will guide a developer into designing a system so that it conforms to well-established prior practices, without stifling innovation.

The patterns have illustrative names and are described with diagrams illustrating their role players. There are only 23 classic patterns (fewer than the letters of the English alphabet), and a good programmer can learn the names and uses of all of them with some practice. When faced with design choices, such programmers are no longer left to select language features such as inheritance, interfaces, or delegates but can instead home in on the bigger picture. They would be able to recognize that an Observer pattern would be suitable for a blog system, and a Proxy pattern would be useful in a community network system. The element of decision-making is not removed, but it is raised to a higher level.

This level of abstraction is a result of combining language features in a tried and tested way. The familiarity of the classic design patterns has contributed much to their success. Design patterns exist also in other areas, such as enterprise systems, security, real-time systems and parallel programming, but their adherents and therefore their impact are confined to a niche community.

Design patterns can be said to be one of the most successful and recognizable abstraction tools that software engineers have at their disposal. However, as mentioned earlier, an abstraction exists in a chain. Design patterns "feed on" programming languages that form a wobbly and constantly moving layer.

2.3 The programming languages layer

Those who have long-term programming experience will appreciate that time brings improvements to a language. Simple things that we take for granted today—like type checking of variables—were nonexistent or optional in the languages of the 1970s. Object orientation, which is the basis for programming these days, only came into vogue in the 1990s, and generics, on which modern collection classes for stacks, maps, and lists are based, were just a research project five years ago.

Java and more notably C# have added significant language features over the last decade. For example, C# 2.0, which was developed between 2002 and 2005, added generics, anonymous methods, iterators, partial and nullable types. C# 3.0, finalized in 2006, focuses on features that would bring the language closer to the data that pours out of databases, enabling its structure to be described and checked more accurately. These features included: implicit typing of local variables and arrays, anonymous types, object and array initializers, extension methods, lambda expressions and query expressions (LINQ).

Successful programmers keep abreast of improvements in languages, but often it is not obvious even to a seasoned professional how a particular new feature will be useful. Some features, such as automatic properties and collection initializers are likely to immediately find a home; others, such as extension methods, are somewhat more abstract. Examples can be used to illustrate the utility of many emerging language features—but while examples illustrate, they can also obscure, because they are directed towards solving particular problems. Given an example of how iterators work with a family tree manager, would it be clear how to reuse them for a chat room program? The connection is not at all obvious and could easily be missed.

[1] As quoted in David Quammen's recent book "Monster of God: the man-eating predator in the jungles of history and the mind", Hutchinson, 2004, from where I got the inspiration for this analogy.

2.4 Pattern implementations

It is still open territory as to whether, and how, new language features should be used in implementing design patterns. In books and writings on web sites the pull of custom is very strong. Because implementations of the patterns were originally given in C++ and Smalltalk, which have their own particular object-oriented styles, the translations into other languages have not always been completely successful. There has been continuing debate over the language features that could make design patterns significantly easier to express [2] [6]. At the same time, the rise of design patterns has coincided with considerable advances in OOPs features in mainline languages such as Java and C#. Concepts such as generics, delegates and iterators, which some of the design patterns were conceived to provide, are now part of the language itself. A debate in 2000 examined the experimental languages of the day (Cecil, Dylan and Self) and looked ahead to first-class generic functions, multiple dispatching and a flexible polymorphic type system [7]. All of these are once again in mainstream languages now.

It is therefore a challenge to make the most of a language, while at the same time retaining the link with the design pattern and its terminology. It is certainly not an aim of design patterns to force a certain way of coding, thus depreciating the value of new language features. Nevertheless, a look at the expository examples in most Java or C# books will show little deviation from the C++ style of the 1990s. It is this "pattern" of complicity that this paper seeks to break.

In [4] I have set out a complete set of implementations of patterns that makes use of novel features of C# all the way up to version 3.0, running on the .NET 3.5 Beta Framework (August 2007). This list of features is shown in Table 1. The list has been sorted according to the features actually required (central column), from simplest (interfaces) to most complex (query expressions).

The features in the central column are those that are absolutely necessary to implement the pattern. However, in order to make a meaningful example, more might be required, and these features are shown in column 3. For example, both the Mediator and Observer patterns can be implemented as console applications, but if they are truly to show their mettle, they will need to run with Windows Forms, in which case threads will be needed to react to events caused by user input.

3. EXAMPLES

3.1 The Bridge pattern

The Bridge pattern decouples an abstraction from its implementation, enabling them to vary independently. As such, it is an interesting pattern to consider in the context of abstraction in the large.

Inheritance is a common way to specify different implementations of an abstraction. However, the implementations are then bound tightly to the abstraction, and it is difficult to modify them independently.

Table 1: Language features used in patterns

Pattern	Language features	Optional and in the examples
Abstract Factory	interfaces	generics, generic constraints
Bridge	interfaces	extension methods
Builder	interfaces	generic, generic constraints
Decorator	interfaces	
Factory Method	interfaces	
Adapter	interfaces, inheritance,	delegates, anonymous functions, threads, events
Proxy	interfaces, private	collections
State	interfaces, selection	
Strategy	interfaces, selection	generics, nullable types
Interpreter	recursion, selection	
Visitor	interfaces, recursion	reflection
Façade	namespaces	
Singleton	private, nested classes, static property	
Template Method	method overriding	
Command	delegates	
Mediator	delegates	threads
Observer	interfaces, delegates, events	threads
Flyweight	interfaces, structs, collections, indexers	implicit typing, initializers, anonymous types
Memento	serialization, collections, indexers	
Prototype	cloning, serialization, collections, indexers	
Chain of Responsibility	generics, exceptions	enumerated types, initializers
Composite	interfaces, collections, generics, properties,	
Iterator	enumerators, foreach, query expressions (Linq)	generics, recursion

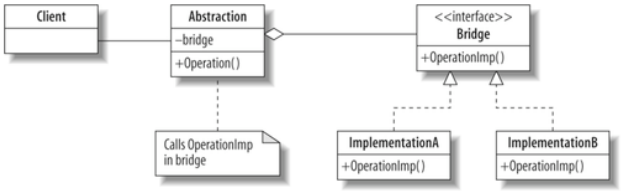

Figure 1. Bridge pattern UML diagram

The Bridge pattern provides an alternative to inheritance when there is more than one version of an abstraction. In the UML diagram (Figure 1) the two implementations, A and B, implement an interface called the Bridge. The Abstraction includes an attribute of type Bridge but is not otherwise in a relationship with the implementations.

Bridge is a very simple, but very powerful pattern. Given a single implementation, we can add a second one together with a Bridge and an Abstraction and achieve considerable generality over the original design. The code related to the diagram is shown in Figure 2.

```
using System;

class Abstraction  {
  Bridge bridge;
  protected internal Bridge Bridge {
    get {return bridge;}}

    public Abstraction (Bridge implementation) {
      bridge = implementation;
    }
    public string Operation () {
      return "Abstraction" +" <<< BRIDGE >>>
      "+bridge.OperationImp();
    }
  }

  interface Bridge {
    string OperationImp();
  }

  class ImplementationA : Bridge {
    public string  OperationImp () {
      return "ImplementationA";
    }
  }

  class ImplementationB : Bridge {
    public string  OperationImp () {
      return "ImplementationB";
    }
  }

class Client {
  static void Main () {
    Console.WriteLine(new Abstraction
      (new ImplementationA()).Operation());
    Console.WriteLine(new Abstraction
      (new ImplementationB()).Operation());
  }
}
/* Output
Abstraction <<< BRIDGE >>>> ImplementationA
Abstraction <<< BRIDGE >>>> ImplementationB
*/
```

Figure 2. Bridge pattern implementation

In this exemplar of the Bridge pattern, the `Abstraction` is implemented in two different ways based on the `Bridge` interface. In addition, the `Abstraction` class on the one hand and the `Bridge` interface on the other can be extended independently. The mechanism remains, but new functionality can be added on both sides. This requires a degree of agreement that the pattern promised to avoid.

If the developer of the `Abstraction` adds further operations, these can optionally be implemented in the classes across the Bridge, since the `Abstraction` is a class not an interface. If however, the implementators want to add common operations to the `Abstraction`, they would need access to it, which they might not have. Enter a new feature of C# 3.0: extension methods.

Extension methods allow developers to add new methods to an existing type without having to create an inherited class or to recompile the original. They therefore play a role at the software design level. An extension method is defined the same way as any other, with two stipulations:

- It is declared as `static` in an outer-level static, nongeneric class.
- The type it is extending is declared as the first parameter, preceded by `this`.

The method can then be called as an instance method on an object of the type that has been extended. To provide `Operation2` without altering or subclassing Abstraction, we add:

```
static class AbstractExtensions {
   public static string Operation2 (
          this Abstraction me) {
     return "Extension <<<BRIDGE>>> "+
            me.Bridge.OperationImp2();
   }
 }
```

This example shows how abstraction (small “a”) can be achieved via patterns, and also how a new language feature can make it easier to achieve over the lifetime of a project.

3.2 Prototype pattern

A different way of raising the abstraction bar is to use library methods. This option is seen in the Prototype pattern (Figure 3). The Prototype pattern creates new objects by cloning one of a few stored prototypes. Objects are usually instantiated from classes that are part of the program. The Prototype pattern presents an alternative route by creating objects from existing prototypes.

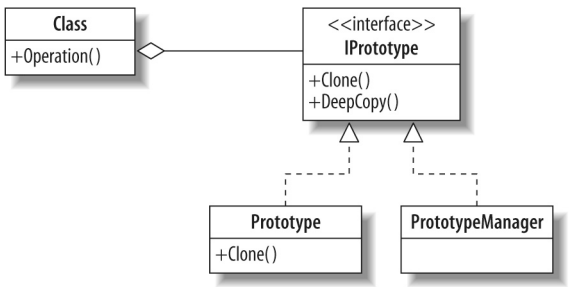

Figure 3. Prototype Pattern UML Diagram

Given a key, the program creates an object of the required type, not by instantiation, but by copying a clean instance of the class. This process of copying, or cloning, can be repeated over and over again. The majority of Prototype implementations stick to "shallow cloning" which works for the copying of a single object. But collections of objects are just as common and cloning a collection requires traversing it, writing it somewhere, and reading it back again. What languages such as Java and C# have done is to implement serialization which can take an arbitrary structure and linearize – or serialize – it. It can also bring it back again. The algorithm for such a traversal of an arbitrary structure is non-trivial, so that serialization is a considerable jump in the abstraction level for patterns. The generic class for serializing a collection in C# is shown in Figure 4.

```
[Serializable()]
 public abstract class IPrototype <T> {

   public T Clone() {
     return (T) this.MemberwiseClone();
   }

  public T DeepCopy() {
    MemoryStream stream = new MemoryStream();
    BinaryFormatter formatter =
           new BinaryFormatter();
    formatter.Serialize(stream, this);
    stream.Seek(0, SeekOrigin.Begin);
    T copy = (T) formatter.Deserialize(stream);
    stream.Close();
    return copy;
   }
}
```

Figure 4. Prototype pattern implementation

With the abstraction provided by the `Serialize` method, a deep copy becomes as simple as shallow cloning, as in Figure 5:

```
[Serializable()]
class PrototypeManager : IPrototype <Prototype>  {
  public Dictionary <string,Prototype> prototypes
    = new Dictionary <string,Prototype> {
    //… details …
  }

static void Main () {

 PrototypeManager manager =
   new PrototypeManager();
 Prototype  c2, c3;
 c2 = manager.prototypes["key1"].Clone();
 c3 = manager.prototypes["key2"].DeepCopy();

}
```

Figure 5. The Prototype Manager and Client

In addition to Serialization, this excerpt shows the use of generics in assisting libraries to be highly reusable. The `PrototypeManager` can declare the specific details of a particular set of prototypes in a dictionary (through a collection initializer) or it could read the data from somewhere.

3.3 The Iterator pattern

The Iterator pattern provides a way of accessing elements of a collection sequentially, without knowing how the collection is structured. As an extension, the pattern allows for filtering elements in a variety of ways as they are generated. The concept of iterators and enumerators (also called generators) has been around for a long time [3]. *Enumerators* are responsible for producing the next element in a sequence defined by certain criteria. The *iterator* is the means by which we cycle through this sequence of elements from beginning to end.

Iterators need data to iterate over. Most of the time the data exists, either in memory, on disk, or somewhere on the Internet. However, sometimes an iterator will work with an enumerator that actually generates values (for example, random numbers). The Iterator pattern is the one that has received the most assistance from language design in recent years. For this reason, the UML diagram in Figure 6 shows the statements that are involved, on an equal footing with the classes.

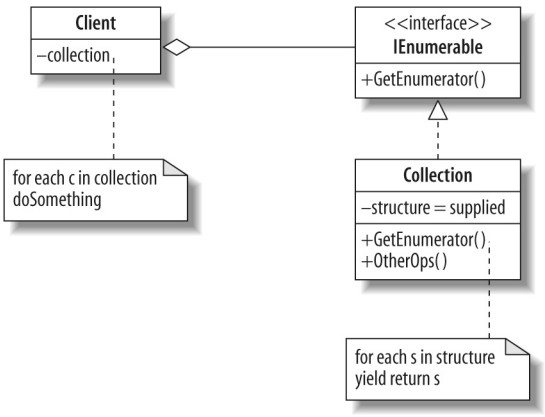

Figure 6. Iterator pattern UML diagram

For many years, iterators were implemented by following an interface that needed the following three methods (Java):

`boolean hasNext()`
: Returns true if the iteration has more elements.

`Object next()`
: Returns the next element in the iteration.

`void remove()`
: Removes from the underlying collection the last element returned by the iterator (optional operation).

Over the past five years, developments in language and compiler technology have wrought a revolution in loop programming. Iteration over any collection of any data type is now linked to a more compact, yet versatile, type-specific enumerator. The inspiration for this new way of thinking comes from the database world (SQL) as well as the functional world (Haskell and Mondrian). Consider the following C# 3.0 statements in Figure 7:

The first statement asks for a selection of files from a collection called `mydir`, with the conditions that they must be dated 2007 or later and that the selection, when made, must be ordered by filename. The statement will work no matter what the type of the collection holding `mydir` is: it could be an array, a linked list, a binary tree, or even a database. This statement works together with an enumerator that:

```
var selection = from f in mydir
   where f.Date.Year >= 2007
   orderby f.Name
   select f;

foreach (var f in selection) {
 Console.Write(f + " ");
}
```

Figure 7. LINQ query syntax

- Follows the structure of the data type in a prescribed order
- Applies the conditions and ordering as specified
- Operates in a lazy fashion, generating values only when they are needed by an iterator.

`IEnumerable` is an interface that has one method, `GetEnumerator`. Classes that implement `IEnumerable` supply a `GetEnumerator` method that successively returns a value from a collection using `yield return`. Each time `GetEnumerator` is invoked, control picks up after the yield return statement.

All collections in the .NET library implement `IEnumerable` (i.e., they each provide a conforming `GetEnumerator` method). That is why they can be used in the foreach statement directly. When `yield return` appears in a foreach loop, it is making use of the underlying `GetEnumerator` for the collection it is iterating over, as in:

```
public IEnumerator GetEnumerator ( ) {
  foreach ( string element in months )
  yield return element;
}
```

In this case, the enumerator for an array (used by months) is invoked. `yield return` keeps the enumerator going for the next iteration. However, a `yield break` statement has the effect of terminating the loop that called the enumerator. `IEnumerator` is a lower-level interface that specifies the `Current` property and the `Rest` and `MoveNext` methods. It is possible to implement this method and thereby satisfy the requirements of the `IEnumerable` interface.

Enumerators can also provide filters, transformations, and projections on the data that can be linked back to the iterators that use them. The enumerators provide methods that do the work, and the iterators can either call the methods directly or use the special LINQ query expression syntax (Figure 7). Query expressions provide a language-integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL and XQuery. The current LINQ syntax developed from lambda expressions as can be seen by considering the equivalent of the selection in Figure 6, minus the sort:

```
// in LINQ
var selection = from f in mydir
   where f.Date.Year >= 2007
   select f;

// as a lambda expression
var selection = mydir.
    Where(f => f.Date.Year >= 2007));
```

Thus, from the abstraction point of view, query expressions present library functions with a considerable dose of syntactic sugar.

4. ASSESSMENT

As we have seen, a design pattern is a formal mechanism of documenting solutions to reoccurring software design problems. The academic and commercial interest in software design patterns has grown dramatically over the last few years, as they are seen as solutions to software design issues. They are, of course, not the only solution to software design, and they should not be used to the exclusion of all others. Component-based design, software architecture, aspect oriented programming, and refactoring also have a place. Viewed against the larger backdrop of software engineering, design patterns can be seen to present some of their own abstraction challenges:

Traceability The traceability of a design pattern is hard to maintain when programming languages offer poor support for the underlying patterns (akin to a large animal struggling to survive when its feeder animals mutate.) The physical implementation of a design pattern in a programming language can be scattered across a number of classes and thus hard to trace (a similar scenario to animals scattering in a drought.) In this respect, the implementations in [4] have made considerable strides by using some of the more compact features of C#, such as delegates and query expressions, which are easier to spot and track.

Reusability Design patterns are used and reused in the design of a software system, but with little or no language support, developers must implement the patterns again and again in a physical programming language. A design pattern does not give a developer the same benefits that a component does, which can encapsulate behavior and be reused as is. Raising the language bar through syntactically sugared libraries can assist in meeting this challenge, as illustrated with the Iterator pattern.

Writability Some design patterns have several methods with trivial behavior. Without good programming tools, it can be tedious to write all this code and maintain it. Once again, more compact and powerful language features can alleviate the programmer's burden.

Maintainability Using multiple patterns can lead to a large cluster of mutually dependent classes, which lead to maintainability problems when implemented in a traditional object-oriented programming language. Current research is investigating how to transform design patterns into reusable artifacts so that developers won't have to implement the same patterns over and over [Meyer and Arnout 2006]. In the context of design patterns, a specific language feature, a pattern library, or a component could solve the pattern implementation reusability problem.

5. CONCLUSIONS

This paper has positioned design patterns as key to the realization of abstraction in multi-layered software engineering. It argues that design patterns can be implemented in many ways, and that the higher the level of the language features that is used, the easier and more understandable are the implementations of the patterns. Considerable progress has been made in raising the bar for language features in C# 3.0, and examples of how the features can be used are explained. These features include both new syntax as well as library methods.

Further work still has to be done on the precise evaluation of the new pattern implementations, as summarized in Table 1. In addition, the opportunity for the reuse of patterns is an area for further research.

ACKNOWLEDGMENTS

Thanks to Alistair van Leeuwen for his assistance with the Assessment section.

REFERENCES

[1] Arnout, Karine and Bertrand Meyer. Pattern componentization: the factory example, Innovations in Systems and Software Technology: a NASA Journal 2 (2) July 2006, 65–79.

[2] Baumgartner, Gerald, Konstantin Läufer and Vincent F Russo. On the interaction of object-oriented design patterns and programming languages, Technical Report, CSR-TR-96-020, Purdue University, 1996.

[3] Bishop, J M. The effect of data abstraction on loop programming techniques, IEEE Trans. on Software Engineering, 16 (4) April 1990, 389-402.

[4] Bishop, Judith. C# 3.0 Design Patterns, O'Reilly, Sebastapol, CA, 2008

[5] Bosch, Jan. Design Patterns & Frameworks: On the Issue of Language Support Proc. Workshop on Language Support for Design Patterns and Object-Oriented Frameworks (LSDF), ECOOP 1997, 133–136.

[6] Bosch, Jan. Design Patterns as Language Constructs, Journal of Object-Oriented Programming 11 (2) 1998, 18–32.

[7] Chambers, Craig, Bill Harrison, and John Vlissides. A Debate on Language and Tool Support for Design Patterns Proc. 27th ACM SIGPLAN-SIGACT POPL Symposium, 2000, 277–289.

[8] Elton, Charles S. Animal Ecology, 1st edn 1927, Sidgwick and Jackson, London. Reprinted several times, e.g. 2001 by The University of Chicago Press, ISBN 0-226-20639-4

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

[10] Gil, Joseph and David H Lorenz. Design Patterns vs. Language Design, Proc. Workshop on Language Support for Design Patterns and Object-Oriented Frameworks (LSDF), ECOOP 1997, 108–111.

[11] Hannemann, Jan, and Gregor Kiczales. Design Pattern Implementation in Java and AspectJ. OOPSLA 2002, 161-173.

[12] Kramer, Jeff and Orit Hazzan. Summary of an ICSE 2006 Workshop: the role of abstraction in Software Engineering, ACM SIGSOFT Software Engineering Notes, 31 (6) November 2006, 38-39.

[13] Kramer, Jeff. Is Abstraction the Key to Computing?, CACM 50 (4) April 2007, 37-42.

[14] Meyer, Bertrand and Karine Arnout. Componentization: the Visitor example, Computer 39 (7) July 2006, 23–30.

[15] Rugaber, Spencer. Cataloguing design abstractions, ROA '06, 11-17, Shanghai, China, May 2006.