The Comparative Pattern System

(formerly entitled **The Quantitative Pattern System**)

Paul Adamczyk
Motorola, Inc.
Arlington Heights, IL
adamczyk@cig.mot.com

Atef Bader
Lucent Technologies
Naperville, IL
abader@lucent.com

Tzilla Elrad
Computer Science Dept
Illinois Institute of Technology
Chicago, IL 60616

Abstract
This paper presents a method of combining various software design and architectural patterns into one pattern system. The software patterns in the Comparative Pattern System are indexed and analyzed from the perspective of the generic design problems that they solve. Our work demonstrates that it is more practical to compare patterns based on how they solve specific problems, rather than on other perceived similarities, such as their scope, type, structure, or problem category. Software patterns document the best architectural and design practices, but their inappropriate application may produce software systems that are hard to comprehend, maintain, and evolve. This paper presents a practical approach that software engineers can apply to determine the optimal pattern for their needs by comparing available alternatives.

Keywords: indexing problem, design patterns, pattern system.

INTRODUCTION

Design patterns were popularized by the so-called "Gang of Four" (GoF) in their book *Design Patterns – Elements of Reusable Object-Oriented Software* [GoF95]. It was the first catalogue describing well-known design solutions using the format of design patterns. It was soon followed by *Pattern-Oriented Software Architecture* book [Bus96] commonly referred to as POSA[1]. This work extended the scope of patterns by incorporating larger-scale architectural patterns as well as low-level patterns, called idioms.

Both catalogues have similar format. Each pattern describes a generic solution of a problem (or a set of problems). The description also includes a short, informal comparison (such as "pattern X uses pattern Y") with other related patterns. Both catalogues also provide a more systematic analysis of pattern relationships by categorizing patterns into groups based on a specific set of criteria. One of the purposes of such classification is to simplify the selection of the best pattern to solve a specific problem. But neither scheme provides a simple index between patterns and problems they can solve.

[1] With the publication of [Sch00], there are two POSA books, so POSA and POSA1 usually refer to [Bus96], while [Sch00] is referred to as POSA2.

GoF classification scheme is very simple. Each pattern is assigned to one of three categories (creational, structural, or behavioral) based on its main purpose. But even some of the patterns described by GoF do not fit clearly into these categories (e.g. Visitor and Singleton). Moreover (as noted in POSA, [Bus96, 367]), the pattern classification based on the patterns presented in GoF is not extensible. It cannot be used to describe some other patterns, such as architectural patterns and idioms.

The scheme proposed in POSA is more flexible than GoF, because it classifies each pattern into a problem category. As new patterns are added, new problem categories can be created for them. One major drawback of this solution is that they classify each pattern to only one problem category[2]. This is an artificial limitation, because each pattern, by definition, must be applicable to three or more different domains. If a pattern applies to many unrelated domains, it is very likely that it can also be used to solve seemingly unrelated problems, but the pattern classification in POSA does not take that into account.

The second limitation of the pattern system in POSA is that it introduces separate problem categories per pattern types, i.e. architectural patterns and design patterns have separate categories. As a result, architectural patterns, which solve distributed problems (Broker, Pipes and Filters, and Microkernel) belong to one category, while design patterns, which deal with inter-process communication (Forwarder-Receiver and Client-Dispatcher-Server) are grouped in a separate communication category. From the practitioner's perspective these two problem categories are similar. Forwarder-Receiver describes how to model peer-to-peer communication, where peers sometimes act as clients and other times as servers. Client-Dispatcher-Server extends Forwarder-Receiver by adding dynamic dispatching capabilities applicable to the client-server architecture. Finally, Broker shows how to extend both patterns to build distributed client-server architecture. A reader interested in finding a specific solution that models client-server architecture must either learn about all patterns from two different problem categories or decide up front what is the scope of his/her problem and select the appropriate pattern type.

THE COMPARATIVE PATTERN SYSTEM

Comparative Pattern System described in this paper provides an alternative scheme of describing relationships between patterns. One of its goals is to addresses the two limitations of POSA classification described in the Introduction. Another goal is to include many patterns from different sources (all domain-independent design patterns) to maximize the possibility of finding the most appropriate solution for a design problem. As the name suggests, the main focus of the Comparative Pattern System is to facilitate comparison between patterns. Our pattern system uses design problems as the basis of the comparison.

This pattern system does not introduce any new design patterns. It uses known patterns and combines them into groups based on specific problems a pattern can solve.

[2] Two exceptions to this are Microkernel and Pipes and Filters architectural patterns, which belong to two distinct problem categories. But this is an exception rather than a rule.

Unlike in the other schemes, one pattern can be listed as a solution of many different problems. In fact, most patterns fit this description.

Observations about Patterns

In some cases, the only reason to consider similarities between two completely unrelated patterns is the fact that they provide a solution to the problem. As long as patterns are compared only based on similarity criteria (such as problem categories in POSA, scope, type, or structure), the only way to recognize that two different patterns are related is to study them and to try to apply them to specific problems. This is a very tedious and time-consuming process, especially since some patterns seem to provide solutions to many general problems.

For example, the Command pattern can be used to:

- model commands as objects
- schedule commands based on their priorities
- provide undo and redo capabilities (when combined with Memento)
- provide logging of activities
- combine primitive operations into transactions

All these uses apply to different kinds of problems and make Command a very versatile pattern. Note however, that for each specific use of Command, the actual implementation of the pattern could be different.

Façade is another pattern that can be applied to a wide variety of problems, but it solves all of them with the same solution – by encapsulating a group of classes behind one simplified interface. This solution can be applied when a set of classes cannot be altered conveniently, to simplify interactions between objects, or instead of redesigning a single class that has become too large.

In some instances, patterns can (and should) be used outside of their typical area. Consider for instance Singleton, which by definition ensures that only one instance of a particular class exists. It can also be used to enforce any other maximum number of instances of a class. GoF describes such a use of the pattern as one of its variants, but this is not its typical usage. A first-time reader of GoF book is not very likely to remember names and basic implementations of all 23 patterns, so it is almost impossible to expect that he/she will recall subtle variants, which often make the difference between obscure and popular patterns.

The advantage of using Comparative Pattern System is that once a pattern is analyzed and is described as a solution of a specific problem, the association between the pattern and the problem is established permanently. This allows for the constant growth of the system without requiring the reader to recall all aspects of all patterns described in the pattern system.

Applicability

All problems described by the Comparative Pattern System are generic design problems. They can be applied in most known domains of software design, because they are domain-independent. It is not necessary for the reader to know anything more than a problem at hand before being able to use the Comparative Pattern System.

This paper analyzes a cross-section of seemingly unrelated patterns from different sources and different categories. All known architectural patterns and design patterns that can be used to solve a given problem are listed side by side as equally valid solutions. Note that this pattern system focuses mostly on design patterns. All problems suggest the use of design patterns rather than any other category. Architectural patterns (and sparingly idioms[3]) are used whenever possible, but they mostly serve as complements to design patterns. They are included to show a wide variety of solutions from different levels of abstraction that can potentially solve the same kind of problem.

The Comparative Pattern System consists of patterns described in four well-known catalogues: GoF, POSA [Bus96], *Pattern-Oriented Software Architecture, Volume 2*, [Sch00] (a.k.a. POSA2), and Grand's *Patterns in Java* [Gra98]. It also includes other patterns, but these four catalogues are its current basis. There exists an extensive number of other catalogues and patterns (as indicated by *The Pattern Almanac 2000* [Ris00]), which are not incorporated in this pattern system yet.

All GoF and POSA1 patterns are incorporated as solutions of generic software engineering problems, but not all patterns from other catalogues are listed, because some of them are not generic enough to solve domain-independent problems.

Motivating Example – Interest Calculation

The example below describes a simplified problem, where the Comparative Pattern System can be applied successfully, yielding an extensive comparison of several patterns.

The example is divided into two sections: "Description of the Problem" and "Solutions and their Comparison." The latter section introduces the problem, while the former develops and analyzes possible solutions.

Description of the Problem

Consider an application, which calculates earned interest in a banking system. Depending on the type of account, we can calculate interest using different rules, but usually at the same time intervals, such as monthly. For savings account, first we need to deduct the monthly fees (e.g. account fee, direct deposit fee, Internet access fee) and accidental fees (e.g. for low balance, too many withdrawals). Then we need to calculate days when balance was above the limit and calculate the average daily balance for these days. Finally, we can apply savings account interest rate to the days when the balance was over the minimum to obtain the interest earned.

Checking account has its own monthly fees (e.g. ATM[4] fee, second card fee) and accidental fees (e.g. overdraft charge, too many ATM withdrawals). It does not have minimum balance requirement, but it requires a minimum balance to earn interest, so we need to calculate the number of days the checking balance was above the limit and the average daily balance for these days. Then interest on the checking account can be calculated.

---

[3] Idioms are usually language-dependent, which makes them less likely to provide generic solutions.

[4] ATM – Automated Teller Machine

Other accounts (money market, CDs[5], etc) have their own (also complex) algorithms to calculate interest. For money market, for example, different interest rates can be applied based on the account balance.

Solutions and their Comparison

To model this interest calculation example, we can use the **Strategy**[6] pattern, which is used to encapsulate algorithms. As the application is traversing through the list of accounts, based on the account type, it creates a concrete Strategy object and delegates the interest calculation to it.

Another pattern applicable here is **Template Method**, because the same sequence of steps needs to be executed for each account type. In our example: deduct monthly fees, deduct accidental fees, calculate days when balance requirements were met, calculate average daily balance for these, and apply the appropriate interest rate. Each account type can be represented as a subclass that implements all these steps in separate methods. During the interest calculation, steps that were overridden will be executed on the concrete class, the remainder will use the base class' implementation.

A third solution is a combination of these two patterns. Apply Strategy pattern, where the Strategy objects have the same private interface, which obeys the rules of Template Method pattern. Each Strategy subclass, in addition to complying with the public interface, also needs to have the same private interface to implement each step of the algorithm.

There are other patterns, which can be used to solve this problem. Their choice depends on the scope of the problem and many other factors. For example, **Builder** creational pattern could be used to model the solution from the point of view of the total interest earned by each account owner. **Iterator** and **Visitor** can be used to obtain similar results. If both patterns are a viable option, use Iterator, because it is easier to implement.

Alternatively, **Pipes and Filters** architectural pattern (whose main purpose is processing of streams of data) could be applied to model this example in case when the result of interest calculation for one account would affect the calculation for another account held by the same owner. In our example, a negative ending monthly balance of the checking account could result in a withdrawal from the corresponding savings account to bring checking account balance to zero. Solutions provided by Strategy, Template Method, or Builder patterns would not be sufficient in this case, because they model each account separately. Pipes and Filters allows to perform interest calculations in such an order that the interest for the checking account would be calculated first and its result would be "piped" to the savings account interest calculation algorithm. Note that Visitor could also be used to obtain similar result.

This example illustrates how different patterns can solve the same problem differently. Each pattern provides a different solution by exploring a specific type of flexibility.

---

[5] CD – Certificate of Deposit

[6] See Table 1, Indexing Patterns to their Catalogues, for references to patterns mentioned here.

SAMPLE PROBLEMS FROM THE PATTERN SYSTEM

Format of Problem Description

This paper includes sample problems and their corresponding pattern solutions. All problems are described using the same format. Each problem description consists of the following sections:

Problem – title of the problem

Description – explanation of the problem that needs to be solved

Analysis – description of forces influencing the solution of the problem

Solutions – ways in which different patterns solve this problem

Relationships between Solutions – explanation of differences between solutions, their scope, assumptions, or drawbacks

Sample problems

Following sections describe four problems from the Comparative Pattern System with their corresponding solutions, analyzed according to the format described above.

PROBLEM 1. Limited Amount of Memory

Description: All applications strive to limit the memory usage. Saved memory can be used to add new functionality without requiring the users to add more memory to existing systems.

Analysis: It is usually possible to save memory by calculating some information dynamically and it is up to the system designers to determine the time/space tradeoff. To achieve best control over memory usage, implement a policy of monitoring the consumption. There are two basic strategies – limit the number of objects of a class that can exist at a time and share data common to many objects. Both strategies can be combined for maximum memory efficiency (at the cost of higher code complexity and, possibly, slower performance).

Solutions:

- Object Pool: Limit the number of instances of a class by pre-allocating all objects during the application's initialization stage.
- Singleton: Limit the number of instances of a class that can be allocated at one time.
- Flyweight: Introduce sharing of common objects (or parts of common objects).
- Counted Pointer idiom: Ensure that shared memory is de-allocated as soon as it is no longer needed.

Relationships between Solutions:

Surprisingly, most of these patterns can be used together to realize the maximum memory savings. By pre-allocating a specific number of objects, Object Pool effectively limits the number of instances of a class. It also reduces the run-time overhead of allocating and initializing memory for new instances, because all allocation and initialization is done up front.

Use Singleton instead of Object Pool for objects that are allocated at run-time. This puts an upper bound on the number of objects, but does not reduce the overhead of creating and initializing them.

Flyweight can be used with Object Pool or Singleton to reduce memory usage even further by having multiple instances share common data.

For programming languages without automatic memory management, add the reference-counting capability (provided by Counted Pointer idiom) to objects that use Flyweight.

PROBLEM 2. Inability to Access Classes Conveniently

Description: Sometimes it is necessary to update a class that cannot be easily modified. For example, if the code is a part of a commercial library, its users cannot modify it. In other cases, it may be possible to change the code, but a change of one class would result in changes of many subclasses so it would be better to avoid it.

Analysis: In such cases, the implementation cannot be altered, but the interface can. So create a new interface to manipulate these classes without affecting the existing code. This can be achieved by adding a new class to manage the communication between the existing class and its client. Another solution is to extend the interface of the original object.

Solutions:

- Adapter: Create a new class with the expected interface and direct the clients to use the new class instead of the existing class.
- Decorator: Dynamically add additional operations to a class. Same for removing operations.
- Façade: Hide such unalterable classes behind a single object with a simple interface so that these clients need to interact with only one object.
- Visitor: Add a new interface to traverse a class structure so that the structure need not be updated when a new traversing mechanism is added.

Relationships between Solutions:

Adapter and Decorator are applicable to a single class, while Façade can encapsulate large sub-systems (including code written in a different language). Façade does not forbid direct access to the sub-system classes, but it doesn't offer an interface to locate such objects, instead it forwards requests on behalf of the client. Decorator is similar in that aspect, because it handles only the "decorated" part of the request and forwards other requests to the original object. In contrast, the use of an Adapter class prohibits the client from accessing the original class directly.

Unlike the other three patterns, Visitor can only operate on class structures, which are not likely to change. But it provides the best solution in such cases, because it allows addition of any type of traversal without modifying the original class structure at all.

PROBLEM 3. Need to Access Private Data of an Object

Description: Sometimes it is necessary to access private data of an object. One such case is to save the internal state of an object so that it can be restored later (by an undo or a recovery procedure). Another reason is to allow a specific class an access to a subset of the private interface of an object.

Analysis: Direct access to private data of an object should be discouraged in truly object-oriented applications. When it is necessary to relax this restriction, the simplest

solution is to implement a mechanism similar to the friend relationship in C++. But this results in tight coupling between classes and opens internals of an object to the outside world. It is better to delegate this responsibility to a separate class.

Solutions:

- Private Interface: Open a small part of the private interface to be accessible only to a specific outside entity.
- Prototype: Create a new object and initialize its internal state with the information copied from the existing object.
- Memento/Snapshot: Store the internal state of an object outside of it so that it is available to be restored later.

Relationships between Solutions:

These patterns provide different means of accessing private data: controlled access to the private interface, access limited to making a copy of an object, and support of undoes without compromising the data integrity of the object.

Prototype provides the ability to access internal data during object initialization only.

Private Interface ensures that only the object, which exposes its private data, can allow another object to invoke a specific private method on it. A side effect of this solution is that the object, which receives the access to the private interface, cannot invoke any public methods on the original object.

Snapshot extends the solution of Memento, because it describes how to manage all the stored memento instances to accomplish various tasks such as serialization (i.e. saving to a file).

PROBLEM 4. Need To Add New Functionality Dynamically

Description: Upgrading an application with a new functionality requires stopping the running program and replacing it with a new version. It is not possible to add new functionality "on the run."

Analysis: There exist techniques to makes software "self-aware" and capable of change at run-time. Just like dynamically linked libraries allow adding new modules to a running application, there are patterns that allow configuring additional functionality to an application that is already running, but only at specific, predefined points.

Solutions:

- Reflection: Create a very flexible, self-learning architecture which supports adding new data, new functionality, and new classes dynamically.
- Interceptor: Create a framework that allows transparent addition of services that are triggered when certain events occur.
- Component Configurator: Allow an application to link and unlink its components at run-time.

Relationships between Solutions:

Reflection and Interceptor are two architectural patterns. Reflection is less dynamic and focuses mostly on collecting information about the application in order to perform services such as serialization of data (to and from the disk). Interceptor allows

adding new services transparently and triggering them automatically when certain events occur.

Component Configurator is a design pattern used to register and de-register services. Interceptor uses Component Configurator to implement this particular part of its architecture. But Component Configurator can be implemented without Interceptor.

INTENDED USAGE OF THE COMPARATIVE PATTERN SYSTEM

The pattern system described in this paper, serves as the index for locating all the patterns, which are applicable to solve a specific problem. Table 1, "Indexing Patterns to their Catalogues," summarizes the problems described in the previous section. It lists the problems, with their respective solutions as well as the source catalogues, which describe each pattern. The reader can use the table as well as the description of each problem to identify design patterns applicable to a specific problem.

The authors envision two main uses of the Comparative Pattern System. First is solving specific design problems. A reader with a problem to solve can analyze the list of all applicable patterns and can proceed to locate and study them. The reader can choose any pattern he/she deems best to solve the specific instance of the problem. The final decision is left to the reader, who is most knowledgeable about the specific context of the problem. The pattern system only suggests all solutions to consider and compares them. The main advantage of using this pattern system is the amount of time saved and the amount of information provided for each known problem. The reader is also spared the trouble of studying patterns, which are not likely to apply to the problem at hand.

The second use of this pattern system is as a guide for studying design patterns. It provides a logical alternative to reading patterns in the order they are listed in their respective catalogues. Thus the reader can study and compare patterns from various, unrelated catalogues concurrently.

Table 1, Indexing Patterns to their Catalogues, summarizes all problems described in this paper and provides references to original sources of patterns that serve as solutions here. The Comparative Pattern System describes many other problems, including:

- Creating an object by specifying a class explicitly
- Dependence on specific operations/requests
- Slow performance of the system
- Non-flexible algorithms
- Complex interactions between objects
- Unmanageably large classes

RELATED WORK

Re-categorization of patterns is not a new idea. Many researchers have attempted to re-categorize patterns, especially patterns described in GoF. Two interesting attempts are [Zim94] and [Alg98]. The first paper divides GoF patterns into three categories – basic design patterns and techniques, patterns for typical software problems, and patterns specific to an application domain. The result is similar to GoF categorization – three distinct categories of patterns with limited applicability to other catalogues. [Alg98] analyzes GoF patterns to determine which ones describe original ideas, rather than known

Table 1. Indexing Patterns to their Catalogues

Problem	Solution	Catalogue
Algorithmic Dependencies (Motivating Example)	Template Method	GoF, 325 and [Gra98, 377]
	Strategy	GoF, 315 and [Gra98, 371]
	Builder	GoF, 97 and [Gra98, 107]
	Iterator	GoF, 257 and [Gra98, 185]
	Visitor	GoF, 331 and [Gra98, 385]
	Pipes and Filters	POSA1, 53
1. Limited Amount of Memory	Singleton	GoF, 127 and [Gra98, 127]
	Object Pool	[Gra98, 135]
	Flyweight	GoF, 195 and [Gra98, 213]
	Counted Pointer idiom	POSA1, 353 and [Cope96]
2. Inability to Alter Classes Conveniently	Adapter	GoF, 139 and [Gra98, 177]
	Decorator	GoF, 175 and [Gra98, 243]
	Façade	GoF, 185 and [Gra98, 205]
	Visitor	GoF, 331 and [Gra98, 385]
3. Need to Access Private Data of an Object	Private Interface	PLoP97 [New97]
	Prototype	GoF, 117 and [Gra98, 143]
	Memento	GoF, 183
	Snapshot	[Gra98, 329]
4. Need To Add New Functionality Dynamically	Reflection	POSA1, 193
	Interceptor	POSA2, 109
	Component Configurator	POSA2, 75

solutions. The results are that about half of GoF patterns are not new ideas. If there existed only 12 patterns, selecting the best one to solve a given problem would be trivial. But neither paper focuses their analysis on pattern selection criteria, which is the main goal of this work.

Another popular strategy of dealing with the increasing number of known design patterns is to attempt to formalize them. Numerous attempts at formal pattern analysis have been published ([Ale95], [Bos98], [Lan97], [Mik97]), but none of them can simplify the process of pattern selection. Structural or semantic similarities between patterns are interesting from the theoretical point of view, but they fail to give software engineers an index to the list of patterns, which can be applied to a specific problem. Moreover, all the papers mentioned above, as well as many others, focus strictly on patterns from GoF and conveniently ignore the additional complexity added by the variety of other known design patterns, architectural patterns, and idioms. The Comparative Pattern System is a practical counterpart of these theoretical works.

For the Comparative Pattern System, POSA and GoF are its two main sources of information and inspiration. Both catalogues describe specific pattern systems.

GoF pattern system is specific to design patterns and since it was the initial catalogue, it does not provide any means of extending it. But it provides an invaluable list of patterns used as a cornerstone of the Comparative Pattern System. The initial idea

for comparing patterns based on specific problems that they can solve was borrowed from this catalogue. GoF lists eight basic design problems, which can be solved by multiple GoF patterns (see [GoF95, 25–26]). Our approach described here uses some (but not all) of the examples described in GoF, with solutions complemented with non-GoF patterns, and adds several different problems and their solutions.

An indirect influence from the GoF book is their use of the same motivating example to describe all creational patterns. Such an approach shows that all creational patterns may be used to solve the same problem. Moreover, it is easier to highlight the key differences between applicable patterns. As shown in "Motivating Example," providing a single motivating example helps highlight key similarities and differences between solutions provided by different patterns.

In the POSA system, relationships between patterns can be shown in three ways – by combining patterns into the same pattern category, by combining them into the same problem category, or by describing it informally in the "See Also" section. Two patterns belong to the same pattern category if they have similar scope, i.e. architectural patterns, design patterns, or idioms. Problem categories are divisions within each pattern category. They combine patterns of similar size providing solutions to related problems, or the same problem. The "See Also" section, in addition to reiterating similarities indicated by the previous two divisions, serves many other purposes. It mentions relationships with non-POSA patterns, provides some indication how patterns from different pattern categories could be related, etc. But neither of these methods conveys the main observation about patterns – that they can solve many unrelated problems. The reader must learn about the pattern, its variants and known uses to recognize some of the areas where given pattern can be applied.

Table of patterns in [Bus96, 380] provides a pattern system combining patterns from GoF and POSA. It also mentions appropriate GoF patterns in the "See Also" section of each pattern. POSA2 extends that idea to include many other patterns. Unfortunately, many patterns in POSA2 are not domain-independent and have limited applicability to this paper.

CONCLUSION

The Comparative Pattern System described above is the next logical step in the analysis of design patterns. As the number of known patterns increases, more sophisticated (yet easier to use) ways of comparing and organizing them need to be developed. Otherwise, the gains from describing new patterns will not be realized by those who need them the most, software engineers. Without simplified indexing, the pattern selection process will become increasingly more difficult.

It is the intention of the authors to analyze more patterns that can solve problems described currently in the Comparative Pattern System. The second goal is to isolate and describe other generic problems that can be solved by many different patterns. In that respect this paper describes just a beginning of a long process. Already mentioned *Pattern Almanac 2000* [Ris00] presents a large list of known patterns that should be included in this scheme. The successful incorporation of the largest number of domain-

independent patterns in this system will show that the Comparative Pattern System is scalable.

Another extension to this work is to develop a more precise differentiation between solutions. Some patterns may only provide a partial solution of a problem. A differentiation between complete and supporting solutions is in order. Also, selection of some patterns may introduce additional drawbacks (e.g. more costly, more complex, or slower implementation). This suggests a need for a more precise cost and benefit analysis of solutions.

Lastly, a web version of the problems described here is also in the list of future expansions of this system.

ACKNOWLEDGEMENTS

The authors would like to thank Jorge L. Ortega Arjona, the shepherd of this paper, for his constructive suggestions and encouragement. Many thanks go also to the members of the Software Architecture and Design group at EuroPLOP for their helpful comments and suggestions.

REFERENCES

[Ale95] Alencar, P. S. C., D. D. Cowan, D. M. German, K. J. Lichtner, C. J. P. Lucena, and L. C. M. Nova, *A Formal Approach to Design Pattern Definition & Application*, Technical Report CS-95-34, University of Waterloo, Waterloo, Ontario, Canada, August 1995.

[Alg98] Algerbo, E. and A. Cornils, *How to preserve the benefits of Design Patterns*, OOPSLA ’98.

[Bos98] Bosch, J., *Design Patterns as Language Constructs*, Journal of Object-Oriented Programming, pp. 18-22, May 1998.

[Bus96] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1. A System of Patterns*, Wiley, New York, NY, 1996.

[Cope96] Coplien, J., *Software Patterns*, SIGS Books and Multimedia, 1996.

[GoF95] Gamma, E, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Gra98] Grand, M. *Patterns in Java, Vol. I*, John Wiley and Sons, 1998.

[Lan97] Lano, K. and S. Goldsack, *Formalising Design Patterns*, Proceedings of 1st BCS-FACS Northern Formal Methods Workshop, 1997.

[Mik98] Mikkonen, T., *Formalizing Design Patterns*, ICSE ’98, Kyoto, Japan, April 1998.

[New97] Newkirk, J., *Private Interface*, Proceedings of PLoP ‘97.

[Ris00] Rising, L., *The Pattern Almanac 2000*, Addison Wesley, 2000.

[Sch00] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Volume 2. Patterns for Concurrent and Networked Objects*, Wiley, New York, NY, 2000.

[Zim94] Zimmer, W., *Relationships between Design Patterns*, Proceedings of PLoP ‘94.