JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 5, September/October 2003

Understanding Symmetry in Object-Oriented Languages

Liping Zhao, UMIST, U.K.
James O. Coplien, North Central College, U.S.A.

Abstract

The success of symmetry applications in many scientific disciplines has motivated us to explore symmetry in software. Our exploration is based on an informal notion that symmetry is the possibility of making a change together with some aspect that is immune to this change. In this view, symmetry has a duality of change and constancy whereby some aspect of an object can be changed while leaving other key aspects invariant. This view of symmetry is a fundamental concept underpinning many symmetry principles in the physical sciences. We have found that we can explain some object-oriented language constructs using this notion of symmetry. This article explores symmetry in object-oriented languages and also provides other examples of symmetry outside of object-oriented programming to show that symmetry considerations broaden beyond object orientation to other areas of software design.

1 INTRODUCTION

The success of symmetry applications in many scientific disciplines has motivated us to explore symmetry in software [7][8][31]. Our exploration is based on an informal notion that symmetry is the possibility of making a change while some aspect remains immune to this change. In this view, symmetry has a duality of change and constancy whereby some aspect of an object can be changed while leaving other key aspects invariant. This view of symmetry has underpinned many principles in the physical sciences. For example, one of the most important conservation laws is that of mass-energy. This law derives from the symmetry principle that physical laws are invariant with respect to time; that is, the amount of energy present before any physical interaction must equal the amount available after the interaction. We have found that this notion of symmetry also underpins some object-oriented language constructs.

This article presents our understanding of symmetry and its role in object-oriented languages. We believe a proper understanding of symmetry is important not only to software design, but also to software maintenance, testing and debugging, as recent

Cite this article as follows: Liping Zhao and James Coplien: "Understanding Symmetry in Object-Oriented Languages", in *Journal of Object Technology*, vol. 2, no. 5, September-October 2003, pp. 123-134. http://www.jot.fm/issues/issue_2003_09/article3

research shows [9]. We shall also provide other examples of symmetry besides object-oriented programming to show that symmetry considerations broaden beyond object orientation to other areas of software design.

The formal notion of symmetry is beyond the scope of this article, but for the completeness, we briefly introduce it here. In mathematics, symmetry is studied under geometry and algebra. Geometric symmetry is a familiar concept in everyday life. It is based on *rigid motion* that leaves distances between points unchanged under some transformation. Such transformation includes *reflection, rotation, translation,* and their various combinations. In algebra [15], symmetry is characterised as a group of transformations that are closed, associative and invertible under a given law of composition. An object possesses symmetry if it remains unchanged under a group of transformations. The basic idea of symmetry is thus invariant change, i.e., change yet the same [21][22][28]. The informal notion of symmetry described above derives from this basic idea.

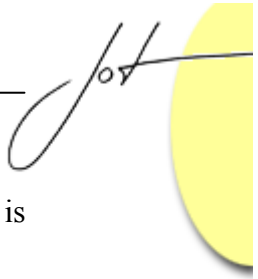
2 SYMMETRY IN OBJECT-ORIENTED LANGUAGES

In this section, we first give an overview of basic object-oriented notions and then discuss how these notions are related to symmetry and what benefits can be obtained from understanding this link.

Basic Object-Oriented Notions

Class is said to be the basic notion of object-oriented programming from which everything else derives [17]. A class is used to describe the structure and behaviour of all the objects generated from the class. Inheritance and subclassing are fundamental relations between pairs of class (and by extension, the transitive closure of these relations). A subclass describes additions (but not deletions) to its direct superclass; it is a convenient mechanism to avoid rewriting definitions that already appear in the superclass. Alternatively, inheritance is the sharing of properties between a class and its subclasses.

Subclasses and inheritance are the basis of *subtyping* [1][5]. Subtyping is a relation on *object types*; an object type with more methods is a *subtype* of one with fewer methods. Consider two objects that share methods m_1, m_2, and m_3, and one of the objects further has method m_4. The type of the second object is a *subtype* of the one with fewer methods. A subtype provides all the behaviour of its supertype and may have extra behaviour. The subtyping relation is reflexive and transitive. *Reflexive* means that a type is a subtype of itself, whereas *transitive* means that if type C is a subtype of type B and B is a subtype of type A, then C is also a subtype of A. Subtyping has a *subsumption* rule, which states that if b is an object of B and B is a subtype of A, b is also an object of A.



The subsumption rule has a practical importance in object-oriented programming and is known as the *Liskov substitution principle* (LSP) [14]:

> *If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2, them S is a subtype of T.*

Classification View of Classes

The class concept itself is interesting in its own right for its close ties to the theory of classification. Several researchers have explored this connection. For example, Wegner [27] presents a classification paradigm for object-oriented programming. He proposes to use equivalence relations as first-order classification mechanisms for classifying elements of a set into independent and disjoint classes. He defines three kinds of equivalence relations as the basis of type soundness and completeness, from which mathematical models of types can be developed.

Motschnig-Pitrik and Mylopoulos [18] study classes and instances from a cognitive science viewpoint. They share Hofstadter's [10] view that our thinking is grounded in our ability to classify, and that classification establishes relationships between classes and instances. They state that classification underlies cognition and serves as an organisational structure for human memory. Cardelli [5] states that classification is what makes object-oriented programming different from other programming paradigms. Object-oriented programming languages generally support two levels of classification, which organises objects into classes and classes into class hierarchies. Simons [23] believes that all the object-oriented concepts can be united in a single theoretical model and explained under a theory of classification. He demonstrates that the notion of class is a first-class mathematical concept. Rayside and Campbell [20] note many interesting similarities between the notion of class in object-oriented programming and the notion of species in Aristotelian logic. They suggest a new way of understanding object-oriented programming, from the viewpoint of biology and taxonomy.

In the following, we take the classification view of classes a step further by studying its link to the general idea of symmetry – the principle of invariance. We shall also explore symmetry in classification with subtyping.

Symmetry in Classes

As just mentioned, a class is a classification of objects. Such classification establishes the class-member invariance such that the description of the class is true for all the objects of the class. This view of class can be explained as symmetry [11]: A class enables the change of the objects, but the change must respect the structure and behaviour stipulated by the class. The structure and behaviour of a class can be enhanced through class invariants, preconditions and postconditions [17]. The benefit of symmetry in classes is

thus that it constrains the change of objects and enforces the correctness of classes. From the compiler's viewpoint, symmetry improves the compilation optimisation. For example, in C++, since all the objects of a class are of the same size and the offsets of all members are the same, they can help the compiler to generate efficient code for member access, method lookup, and memory allocation.

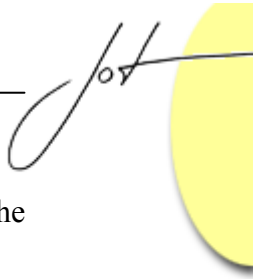
This notion of class can be valuable to component-based software development. Components can be grouped into classes according to certain commonality, such as interface compatibility or behaviour compatibility. A class can then be used to generate all the components that have the same commonality and accommodate individual variations. The components of the same class thus provide the flexibility for change. When a requirement is changed in an application, a new component can be used to replace the old one so long the new component respects the defined commonality in the component class. Such substitutability is an important feature in component-based software development. It is also important to software maintenance and evolution where the replacement of old or erroneous components is a major concern.

More importantly, the link between symmetry and classification provides a basis for theoretic study of object-oriented software design. By applying symmetry principles, the aim of software design is to identify and preserve the maximum design invariants and support variations. This aim underlies design reuse. We can envisage such a design method that provides a collection of design templates. Design templates describe invariants and possible variations. Design is then about instantiation of these templates, composition of the instances and substitution of the instances. The general notions of classes and objects can be extended to design templates.

Symmetry in Subtyping

Inheritance has been largely related to generalisation or specialisation in the object-oriented programming [26]; only a few researchers relate it to classification [5][23][27]. In knowledge representation, Winograd [29] perceives inheritance hierarchy as a system of classification where each node in inheritance represents a class of objects, and an *is-a-kind of* link connects a class to some superclass that properly contains it. Winograd states that applying classification is one of the basic modes of reasoning and classification, by its very nature, is hierarchical.

In object-oriented programming, the role of inheritance in classification is not as clear as that of classes. The lack of the clarity owes to the flexibility of the inheritance mechanism. Inheritance can be used to extend a class, restrict a class, or otherwise modify a class. This means that a subclass may not be properly contained within its superclass and the description of a superclass may not be true to all its subclasses. Hence inheritance does not always classify classes. However, when inheritance is used as *subtyping*, as described above, it can be viewed as classification of classes, in that it establishes the behaviour invariance between a subclass and its direct superclass such that the subclass *behaves* the same as its superclass. We wish to make clear that in practice,



subtyping is only one of the many roles inheritance plays. Therefore, it is only under the subtyping relation that inheritance classifies classes.

Subtyping is related to symmetry: all the classes of a subtyping path may vary, but they must preserve and conform to a common behaviour. All the classes of a path through a subtyping structure can be said to conform to the same *type*.

The benefits of understanding subtyping in terms of symmetry are similar to those of classes. First, the subtyping invariants can act as type constraints on inheritance to impose (for example, at compile time) a uniform typing behaviour for all classes in an inheritance path. Subcontracting rules [17] are a way of specifying subtyping invariants. Second, the notion of subtyping can be applied to component-based software development to achieve the desired substitutability among the components. Finally, classes and subtyping are basic design mechanisms that support the change through the substitution and maintain the stability through the invariants.

Symmetry in Operator Overloading

One of the reasons for providing the operator-overloading feature in the C++ programming language at the time was that it "looked neat" ([25], p.78). Another reason was to prevent some operators from being member functions with full access to the class private members. There were also symmetry considerations for overloading operators [13]. For example, by overloading an operator, one can treat it as an arithmetic operator and supplies the left and right operands to this operator. However, strictly speaking, overloaded operators do not provide symmetry of left and right; they only create an illusion of it, because overloaded operators such as "+" and "*" are not commutative and do not therefore possess reflection symmetry. In addition, the invocation of the overloaded operator at run time has little or no symmetry because in simple object-oriented programming, the language chooses (at run time) from among implementations of a polymorphic operations based on the (dynamic) type of *one* of the objects involved in the operation. It is only statically that the overloaded operator *looks* symmetric with respect to its left and right operands.

However, we can imagine a *historical symmetry* exists that preserves the structure and uniformity of the arithmetic operators, familiar from common algebra. Hence the importance of the operator-overloading feature is that it provides symmetry between built-in data types and user-defined types.

Symmetry in Double Dispatch

Double dispatch is a mechanism that associates a method with two objects and implements multiple methods. Double dispatch can happen at compile time (static dispatch) or runtime (dynamic dispatch). Double dispatch is a way of supporting polymorphic method lookup that goes beyond the simple dynamic single-object type

lookup of Smalltalk, Java or C++, but which recalls the more powerful but statically bound polymorphism of arithmetic operators in, say, FORTRAN.

Overloading is a simple form of polymorphism, a simple double dispatch. In the previous section, we showed its relation to symmetry.

Whereas opinions divide on whether or not double dispatch should be used in object-oriented programming, the recent development shows the importance of symmetry in double dispatch [2]. When two arguments of the multimethods have the same type or conform to the same type as in subtyping, a symmetric multimethod can be added to make the order of the two arguments insensitive. An obvious benefit of symmetric dispatch is that the client code does not need to remember the order of arguments. Such feature simplifies the use of multimethods.

Symmetry also appears in the `equal` method of the Java programming language. The contract of the `equal` method requires that any implementation of `equal` should be symmetric, so that `a.equal(b)` should always produce the same result as `b.equal(a)`. The importance of this symmetry requirement is to simplify the coding and to constrain the behaviour of the code.

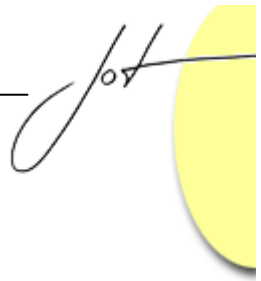
3 OTHER SYMMETRY EXAMPLES

The above section serves to illustrate symmetry in object-oriented languages. In this section, we give more examples of symmetry. Although some of the examples are tenuous visual or geometric symmetries, a good understanding of them is just as important.

Symmetry in Program Format

The simplest form of geometric symmetry in software is the format of programs. A well-formatted or indented program can be understood more easily than a poorly formatted or indented one. While there is no agreement on what a good formatting should be (e.g., there are endless arguments about whether open curly braces should be on a line by themselves or at the end of the line preceding the block), there is a basic symmetry pattern that underlies formatting. In [6], Coplien provides examples of different formatting styles and shows that reflection symmetry of up and down gives a "good shape" to programs. Two such examples are given in Figure 1. Although it may sound like a superficial or even trite finding, there is anecdotal evidence from projects in Bell Laboratories that consistent indentation style is a good indicator for low bug density in code.

```
int gilligan(int j) {
    for (int i = 1; i < j; i++)  {
        if (i % 2 == 0)  {
            count << i;
```



```
        }
    }
}

procedure playBallGame()
    begin
    char key;
    integer ballsLeft;
    procedure playABall()
        begin
        integer count;
        procedure checkPosition();
            begin
            integer x, y;
            y := ball.yposition;
            . . . .
            end
        . . . .
        end
    end
```

Figure 1: Up-down Reflectional Symmetry in Code

Symmetry in Software Development Life Cycles

Raccoon [19] presents a chaos model for the life cycles of software development. The chaos model uses fractals to represent different phases of a life cycle such that the complete life cycle can be interpreted in terms of each phase, or conversely, each phase can be interpreted in terms of a complete life cycle. Fractals are a special kind of symmetry – the symmetry of self-similarities [16]. The purpose of using the principles of fractals as a metaphor in describing software development phases is, according to Raccoon, to bridge the gap in our understanding between one line of code and the entire project, because the software development processes are recursive, similar and scalable. Figure 2 shows the chaos model that combines a linear problem-solving loop with fractals to describe the complexity of software development. Raccoon explains that fractals imply a very specific scaling relationship in the recursion. The same expansion applies to each level. All levels of software development have the same value to the project as a whole. Each level is composed of all the levels below it and so each level repeats the structure of the next level down. Raccoon demonstrates that the chaos model has unified many facets of software development and can lead to a better understanding of software development. We find similar symmetry in more widespread life-cycle models such as Boehm's spiral model [4].

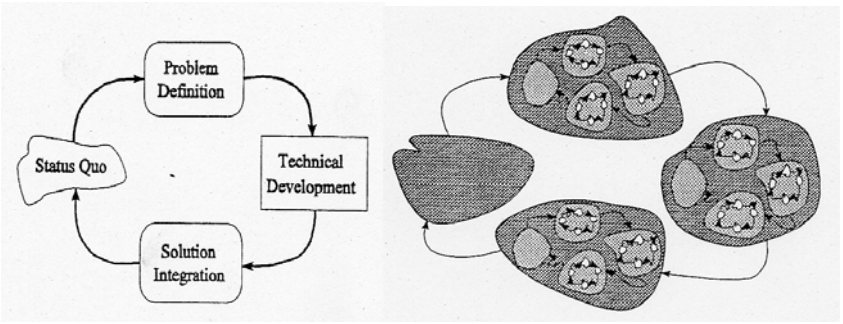


Figure 2: Fractal symmetry in Software Development Processes (after [19]).

Symmetries arise in many design problems. In the design of public transport systems [30], we note that the driver duty components are symmetric to their builders. A driver duty can be seen as a translation of its builder. Like the chaos model, the role of symmetry in here is also to preserve the similarity of the structure.

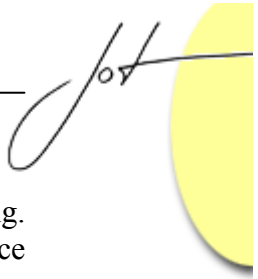
Symmetry in Search Algorithms

Symmetry has found to be most successful in assisting the search of solutions for combinatorial design problems in constraint programming [24]. Combinatorial problems are in general NP-complete. They are typically solved as constraint satisfaction problems by assignment search algorithms. Symmetries in these problems can lead to redundant search, where many symmetrically equivalent blind alleys are explored wastefully. Symmetries divide the set of possible assignments into equivalence classes. Each class contains either only solutions or no solutions. According to this feature, search algorithms can adopt various strategies to handle the symmetries. For example, if a search algorithm has found that an assignment will not lead to a solution, it will abandon search for the symmetric assignments to the assignment already considered. If, on the other hand, the search algorithm has found a solution from an assignment, it will also abandon search for the symmetric solutions. Smith [24] describes a symmetry breaking strategy that reduces symmetries by adding constraints during the search to ensure that any assignment symmetric to one already considered will not be explored.

4 CONCLUSION

The essence of symmetry is invariant change. This article explores this understanding of symmetry in object-oriented languages and the benefits of symmetry. In general, symmetry in software can be viewed as an invariant change that aims to preserve a specific property of a system, which is summarised as follows:

1. *Structure*. An example of structural preservation is the notion of class. A class stipulates a uniform structure for all the objects of the class.



2. *Behaviour*. An example of behavioural preservation is the notion of subtyping. Subtyping stipulates a uniform behaviour for all the classes along an inheritance path.
3. *Regularity*. Symmetry considerations in operator overloading and double dispatch have the effect of preserving certain regularity. It is also true to say that structural and behavioural preservation is also about protecting regularity.
4. *Similarity*. Similarity is a comparison between two things. When the two things are similar, their similarity is captured as something regular. The chaos design model exhibits such regularity.
5. *Familiarity*. When something is familiar, it has the similarity of something else. Preserving familiarity can be called historical symmetry, symmetry of the past and the present, primitive types and user defined types.
6. *Uniformity*. This is just a different way of seeing regularity.

In this article, we have illustrated these properties with the examples of classes, subtyping, program formatting, operator overloading, double dispatch and chaos design model. These properties are related – they are just different manifestations and effects of symmetry. They can be explained from one single viewpoint: they simplify the design and preserve the regularity. It is worth noting that this view of symmetry is consistent with those of gestalt psychology [12] and information theories [3] and has the importance in general human understanding, far beyond the software arena. We shall leave this further exploration of symmetry to another paper.

This paper has also used a search algorithm example to illustrate that the presence of symmetry is not always a good thing, as symmetry provides redundant information that can lead to redundant search. However, knowing such side effect is also to our advantage for we can reduce the search space by breaking symmetry. We have discussed symmetry breaking in software design elsewhere [7][8][31]. For now we conclude that symmetry is a powerful concept that can produce *good effects* on programming and software. However, good symmetry effects can only be obtained based on a good understanding of symmetry, for which we need a conscious effort.

5 ACKNOWLEDGEMENTS

Darius Zakrzewski has provided detailed and insightful comments on the paper. We are grateful to his continous support and encouragement to our symmetry work.

REFERENCES

[1] Abadi, M. and Cardelli, L. *A Theory of Objects*. Springer, 1996.

[2] Alexandrescu, A. *Modern C++ Design*. Addison-Wesley, 2001.

[3] Attneave, F. "Symmetry, Information, and Memory for Pattern". *The American Journal of Psychology* 68(2): 209-22, 1955.

[4] Boehm, B. *Software Engineering Economics*. Prentice-Hall, 1981.

[5] Cardelli, L. "A Semantics of Multiple Inheritance". In *Semantics of Data Types, LNCS* 173: 51-68. Springer-Verlag, 1984.

[6] Coplien, J. "Space: The Final Frontier". *C++ Report* 10(3). New York: SIGS Publications, March 1998, 11-17.

[7] Coplien, J. "The Future of Language: Symmetry or Broken Symmetry?" In *Proceedings of VS Live 2001*, San Francisco, California, January 2001.

[8] Coplien, J.O. and Zhao, L. "Symmetry Breaking in Software Patterns". In *Lecture Notes in Computer Science Series*, LNCS 2177, Springer, October 2001.

[9] Godefroid, P. and Prasad, S. "Symmetry and Reduced Symmetry" in Model Checking. *Proceedings of CAV'2001 (13th Conference on Computer Aided Verification)*, Paris, July 2001.

[10] Hofstadter, D.R. *Godel, Escher, and Bach: An Eternal Golden Braid*. Penguin Books, 1979.

[11] Jensen, W.B. "Classification, symmetry and the periodic table". *In Symmetry: Unifying Human Understanding*. I. Hargittai (ed), Pergamon Press, Oxford, 1986. ISBN 0-08-033986-7.

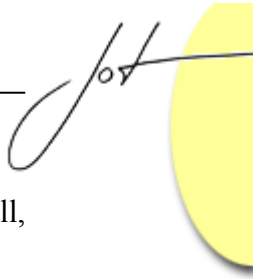
[12] Koffka, K. *Principles of Gestalt Psychology*. London: Geo. Routledge & Son Ltd., 1935.

[13] Lippman, S. B. and Lajoie, J. *C++ Primer. 3rd Ed*. Addison-Wesley, 1998.

[14] Liskov, B. "Data Abstraction and Hierarchy". *SIGPLAN Notices* 23,5, May 1988.

[15] Mac Lane, S. and Birkhoff, G. *Algebra*. New York: Chelsea, 1988.

[16] Mandelbrot, B.B. *The Fractal Geometry of Nature*. W. H. Freeman and Company, New York, 1983.



[17] Meyer, B. *Object-Oriented Software Construction*. 2nd Ed., NJ: Prentice-Hall, 1997.

[18] Motschnig-Pitrik, R. and Mylopoulos, J. "Classes and Instances". *International Journal of Intelligent and Cooperative Information Systems*, 1(1): 61-92, 1992.

[19] Raccoon, L.B.S. "The Chaos Model and the Chaos Life Cycle". *Software Engineering Notes*, 20(1): 55-66, January 1995.

[20] Rayside, D. and Campbell, G.T. "An Aristotelian Understanding of Object-Oriented Programming". *OOPSLA '00*. 10/00 Minneapolis, MN, USA.

[21] Rosen, J. *Symmetry in Science: An Introduction to the General Theory*. New York: Springer-Verlag, 1995.

[22] Rosen, J. "Symmetry at foundations of science". *In Symmetry 2: Unifying Human Understanding*. I. Hargittai (ed), Pergamon Press, Oxford, 1989. ISBN 0-08-037237-6.

[23] Simons, A.J.H. *A Language with Class: the Theory of Classification Examplifed in an Object-Oriented Language*, PhD Thesis, University of Sheffield, 1995.

[24] Smith, B.M. "Reducing Symmetry in a Combinatorial Design Problem". University of Leeds, School of Computing, *Research Report Series*, Report 2001.01, January 2001.

[25] Stroustrup, B. *The Design and Evolution of C++*. Addison Wesley, 1994. ISBN 0-201-54330-3.

[26] Taivalsaari, A. "On the Notion of Inheritance". *ACM Computing Surveys*, 28(3): 438 – 479, September 1996.

[27] Wegner, P. "The Object-Oriented Classification Paradigm". In *Research Directions in Object-Oriented Programming*, Shriver, B. and Wegner, P. (eds), The MIT Press, 1987.

[28] Weyl, H. *Symmetry*. Princeton University Press, © 1952.

[29] Winograd, T. "Frame Representations and The Declarative/Procedural Controversy". In *Representation and Understanding: Studies in Cognitive Science*, 185-210, Bobrow, D.G. and Collins, A.M. (eds), New York: Academic Press, 1975.

[30] Zhao, L. and Foster, T. "Driver Duty Constructor: A Pattern for Public Transport Systems". *The Journal of Object-Oriented Programming* 12(2), May 1999, 45-51;77.

[31] Zhao, L. and Coplien, J.O. "Symmetry in Class and Type Hierarchy". In James Noble and John Potter, eds., *Conferences in Research and Practice in Information Technology*, 10. Australian Computer Society, January 2002, pp. 181-190.

About the authors

Liping Zhao is a lecturer at the Department of Computation, UMIST, U.K. Her research interests include object-oriented design, software patterns and symmetry. She can be reached at liping@co.umist.ac.uk.

James O. Coplien is an associate professor at the Department of Computer Science, North Central College in Naperville, Illinois, a visiting professor at UMIST, and is the current Vloebergh Professor of Computer Science at Vrije Universiteit Brussel. He is a well-known author of books and articles about object-oriented programming and patterns. He can be reached at JOCoplien@cs.com.