By Liping Zhao

PATTERNS, SYMMETRY, *and* SYMMETRY BREAKING

*Explaining software patterns by breaking the symmetry
created by programming languages.*

Software design patterns capture tried and successful design solutions [6]. Among different views on design patterns is that they are created to compensate for the design shortfalls in programming languages [5]—that is, design patterns are needed when programming languages cannot do the job in a straightforward way. Based on this view, Coplien and Zhao [5] postulate that there is a causal relationship between language features and design patterns and that relationship is couched in a more fundamental relationship between symmetries and broken symmetries. This article builds on that postulation and provides a further understanding and fresh articulation of patterns, symmetries, and broken symmetries.

ILLUSTRATION BY PEP MONTSERRAT

Most people have an intuitive understanding of symmetry, which often means a well-proportioned, well-balanced image, such as the appearance of the human body. In mathematics, symmetry is an operation (for example, a reflection) that manipulates the image (the human body in this example) rather than the image itself [10]. More accurately, symmetry is a geometric transformation under which the form of the image remains unchanged [10]. For example, reflection is a symmetry transformation under which the human body remains the same. Other symmetry transformations are rotation and translation. Hence a perfect starfish looks the same under five rotations (see Figure 1a), five reflections (see Figure 1b), and any number of translations—shifting along a straight line; a round ball remains the same under any number of rotations and translations. Objects like these, which remain unchanged upon reflection, rotation, or translation, are said to be symmetric or to exhibit symmetry.

Figure 1. (a) The perfect starfish has a 5-fold rotational symmetry, about the axis through the center by 72° , 144° , 216° , 288° , and 360° ; (b) 5-fold reflection symmetry, one for each arm.

There is a close relationship between geometric symmetry and the concept of invariance [12]. Symmetric objects are also said to be invariant, because their geometric features are preserved (unchanged) under symmetry transformations. Almost every conservation law in physics stems from fundamental symmetry in nature or from some basic principle of invariance. For example, the energy conservation law is based on the principle that the total energy, including that bound up as mass, must stay constant after a physical interaction. Hence, a physical interaction is a symmetry operation on the energy, which leaves the amount of energy unchanged.

Symmetry is closely related to isomorphism. From the viewpoint of abstract algebra, a symmetric system is isomorphic with itself; or, phrased differently, a symmetry operation is an isomorphism between two identical configurations of the system. For example, the starfish exhibits pentamerism and is isomorphic with respect to five rotations (see Figure 1a) and five

reflections (see Figure 1b); each of these 10 symmetry operations is an isomorphism between two identical configurations of the starfish. Thus, isomorphism requires that two symmetric systems are not only apparently the same but also are exactly the same. Many programming features can be explained using symmetry either in its general concept of invariance or in its specific form of geometric transformation [5].

SYMMETRY BREAKING

In physics, symmetry breaking happens when a force disturbs a symmetric system and causes it to break its symmetry to maintain its stability. This phenomenon has been explained in great detail by Stewart and Golubitsky [10] with abundant exam-

ples. One example is a splash of a milk drop. When a drop of milk hits the surface of a bowl, its spherical symmetry is broken into a crown (see Figure 2a). Under ideal conditions, a bilaterally symmetric aircraft should fly straight; in reality, however, the aircraft flies in a slightly zigzag way, because the flow of air past the aircraft is not bilaterally symmetric and the aircraft must break its symmetry to maintain its stability. Symmetry breaking is an important concept in many scientific fields. For example, the embryo development in biology and the electroweak interaction in physics are explained as a process of symmetry breaking [10].

Yet symmetry breaking does not mean a total loss of symmetry, only a partial loss. Before the milk drop hits the surface of a bowl it is invariant under all rotations; when hitting the surface, the milk drop breaks into a crown, which is invariant to only some rotations. If a ping-pong ball is compressed with a uniform radial force, it will buckle. Buckling will knock out some, but not all, of its rotational symmetries (see Figure 2b), because it will react to the force to rearrange itself to preserve as many of symmetries as possible to maintain its stability. Therefore, symmetry breaking is relative to symmetry: If the initial system is invariant to symmetry X, then after symmetry breaking, the resulting system is invariant to symmetry Y, such that Y is smaller than X.

Symmetry breaking has been recognized as a fundamental process of pattern formation [10]. If the force on the ping-pong ball is uniform, buckling will produce a symmetric pattern of identical dents on the surface of the ball. Thus, the pattern on the surface of

the ball arises from breaking the perfect symmetry on the ball. Stewart and Golubitsky [10] relate this phenomenon to the human perceptual quality that the human mind perceives too much symmetry as a bland uniformity rather than a striking pattern. Mathematically, as they explain, a uniform, featureless plane has a vast amount of symmetry; but people do not look at a wall painted in a single color and enthuse over its wonderful patterns. Similarly, a pond has a great amount of symmetry on its surface, but we are intrigued by the less symmetric pattern manifested in circular ripples on the pond. Patterns therefore arise from the contrast, the imperfection against the perfection, such as clouds against the blue sky, ripples on a fish pond, Chinese paper cutting, and the blot of ink on a paper. In all these examples, symmetry provides a norm—a backdrop—against which patterns arise.

Alexander, whose work on architectural patterns is a major influence on software patterns, has said that his understanding of patterns is closely related to the idea of symmetry breaking [2]. In [5], it is shown that software patterns can be explained as a result of breaking the symmetry created by programming languages.

SYMMETRY CREATED BY OBJECT-ORIENTED LANGUAGE FEATURES

Traditional OO design suggests that OO programs should be built from many small objects, like Lego bricks. This view has been challenged in [9], which reported there is no evidence of a typical size of objects (Lego bricks) to OO programs, and instead, OO programs are built from objects of variable size that exhibit scale-free, fractal geometry. The fractal geometry of OO programs may have important implications for debugging and garbage collection, according to [9], as the number of outgoing references of objects can be estimated using a power law; it may also help the designers to predict the number of objects and their relative sizes required in an OO program. Perhaps more important, this new revelation of OO program structure has reaffirmed the idea that software design, like architectural design, is essentially geometric.

Figure 2. Symmetry breaking: (a) a splash of milk drop has lost its spherical symmetries to become a crown; (b) a buckled sphere has lost some of its rotational symmetries (from D'Arcy Thompson, *On Growth and Form*, Cambridge University Press, 1972).

This article takes a step further by positing that the fractal geometry of OO programs is full of symmetries, because OO programming languages, especially class-based languages, such as C++, Eiffel, and Java, have a great desire for symmetry enforced by their type systems. Since each type enforces some invariance to its instances, type systems are inherently symmetric. More generally, symmetries in programming languages are introduced by language designers in the name of design consistency, as expressed by Meyer [8]: “Throughout the design of a class library ... we must constantly ask ourselves: ‘How do I make my next design decision compatible with the previous ones?’ ‘How do I take my next design decision so that it will be easy—or at least possible—to make future ones compatible with it?’”

According to Meyer [8], the concern for symmetry is a special case of the concern for consistency. This article, however, views symmetry as a more fundamental design concept than consistency and that the concern for consistency is but a special case of the concern for symmetry. Symmetry reigns in language design. Class-based languages, for example, exhibit class symmetry, behavior symmetry, template symmetry, and method symmetry. This article explores and reveals these symmetries.

Class symmetry is associated with classes. Although it is clear the class concept is central to class-based languages, it is not always clear why the class concept is so important. According to Stroustrup [11], the importance of classes lies in their organizational power, rather than their computational power. He says: “C with classes was explicitly designed to allow better organization of programs; ‘computation’ was considered a problem solved by C [11].” A class specifies how a type of object can be constructed. Each type of object is invariant under its constructors because all the objects of the same type have the same configuration under their constructors. Objects of the same class can be said to be symmetric to each other because they belong to the same type. Classes can be viewed as a classification tool that classifies objects according to type invariants or type symmetries.

The Eiffel programming language imposes more class symmetries than other class-based languages, to the extent that Eiffel classes are too rigid. Every object of an Eiffel class is an invariant with respect to a set of assertions [8]. Eiffel has a strong connection to abstract data types, such that not only every object of

a class is symmetric to its type, but also every class is intended to be isomorphic to the type.

Behavior symmetry is linked to subtyping relation on object types [1]. An object type with more methods is said to be a subtype of the one with fewer methods. Subtyping is governed by the subsumption rule [1], which states that if b is an object of type B and B is a subtype of type A , b is also an object of type A . A practical use of this rule is known as the Liskov Substitution Principle (LSP) [7], stated as follows: “If for each object $o1$ of type S there is an object $o2$ of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when $o1$ is substituted for $o2$, then S is a subtype of T .” The importance of the LSP is that it places a behavioral constraint on subtypes and ensures a subtype object can always be used safely instead of a supertype object. Types that conform to a subtyping relation form a type family or type hierarchy. Such a type hierarchy is homogeneous because all the types in the hierarchy are of the same kind, a classic view of the “is a kind of” (IS-A) relationship.

Such a type hierarchy also has two important consequences. First, its objects have behavior symmetry because they behave exactly the same from an external observer’s viewpoint and can be substituted one for another in terms of the LSP. As long as the LSP is not broken, the type hierarchy will be type-safe. Second, its objects have a polymorphic type because they can be viewed as belonging to many types of the type family. It is worth noting that objects that have class symmetry also have behavior symmetry because they behave exactly the same, but the reverse applies only to the objects belonging to the same type.

Template symmetry is associated with class templates in C++. A class template or generic type specifies how individual classes can be constructed much like the way a class specifies how individual objects can be constructed. Class templates are also designed for better organization of programs. Classes and class templates provide two orthogonal organizations of programs, such that classes organize objects into types, whereas class templates organize types into generic types. Class templates can be viewed as higher-level or meta-level type constructors. They are similar to meta-schemas in relational databases. All the types generated from the same class template have the same configuration and are therefore symmetric.

Class, behavior, and template symmetry can be generally referred to as substitution symmetry, as objects can be substituted one for another under class

and behavior symmetry whereas types can be substituted one for another under template symmetry.

Method symmetry exists in many forms. In the first form, method symmetry is related to subclass methods. As a direct consequence of the subsumption rule, the sound typing rule for method overriding requires that the types of arguments of methods should vary contravariantly from classes to subclasses, and the types of results of methods should vary covariantly [1]. Another form of method symmetry is operator overloading. In C++, arithmetic operators are overloaded in classes to allow objects to behave as if they were primitive data values and to be expressed as the symmetry of left and right operands. Deeper than expressing this reflection symmetry, overloaded operators create symmetry of familiarity between built-in data types and user-defined types. Experienced programmers know the importance of method symmetry and have introduced many ad hoc symmetries, including complementary method pairs. For example, where practical, a creation method may match a disposal method; a getter and a setter; and a producer and a consumer. In Eiffel, every class method is an invariant against its pre/post conditions.

SYMMETRY BREAKING PRODUCES PATTERNS

Symmetry plays a major role in design for its aesthetic and functional value. One function of symmetry is its predictability: Symmetry shows us what to expect. In OO programs, class symmetry helps the compiler predict the behavior of objects, and method symmetry helps predict the behavior of methods. Another function of symmetry is imposing design consistency, as described previously. Yet perfect symmetry is austere in the extreme. In the visual world, our eyes often find a richer aesthetic content on partly symmetric reality. In biology and physics, the imperfections and irregularities in the arrangement of cells are more interesting because they pose deeper questions to scientists.

In program design, too much symmetry can lead to inflexible and rigid programs. Language designers have long recognized this problem and have introduced language features to break the rigid design. For example, abstract classes and interface features are designed to break class symmetry into interface and implementation symmetry. When language features are inadequate for breaking symmetry, software designers will invent their own gadgets to do the job. Over the years, some of these gadgets have survived and become what is now collectively called design patterns.

Design patterns thus viewed, have therefore arisen from breaking symmetry in OO programs. Although

WHILE DESIGN PATTERNS BREAK SYMMETRIES ENFORCED BY ORIGINAL LANGUAGE FEATURES, THEY CREATE THEIR OWN LOCAL SYMMETRIES, AND IN SO DOING, THEY ADD EXTRA FEATURES ON TOP OF LANGUAGES.

they are now used as solutions for breaking symmetry, they were actually the structure configurations left by symmetry breaking. Here, some popular class-based OO design patterns as in [6] are used as examples to illustrate how they break class, behavior, and method symmetry (for a formal definition of pattern, see [5]).

Class symmetry can be broken in many ways. A straightforward class symmetry breaking is using the abstract class or interface feature. When a language has no interface feature, such as C++, patterns, like Handle/Body and Handle/Body Hierarchy [4] (or Bridge [6]), can be used to break class symmetry. Class symmetry breaking plays a major role in the formation of the design patterns presented in [6], as many of those patterns stem from the idea of breaking class symmetry. For example, all the Creational Patterns are based on the idea of separating the object-construction process from the construction interface.

Why is the separation of a class into an interface and an implementation breaking class symmetry? It is because a class is broken into two. The original class symmetry is now split between the interface (where type invariants are found) and the implementation class (where implementation invariants are found). Since there can be more than one implementation class behind an interface, an interface can represent many sets of objects, each of which is symmetric to a specific implementation class. The original class symmetry is thus broken.

Breaking class symmetry also breaks object type

symmetry, as the object type is now transferred to the interface. Moving a type away from its objects, to the interface breaks the type symmetry of these objects, as an object type can now represent more than one set of objects.

Breaking behavior symmetry happens when a type hierarchy breaks and objects in the hierarchy cannot be substituted one for another. Typical patterns that break behavior symmetry are Adapter, Composite, and Decorator [6].

It is easy to understand that the Adapter pattern breaks behavior symmetry because it transforms one type of object into another type. Suppose we have a list and wish to use it as a set. Since a list is not a set, the two cannot replace one another. The only way we can use a list for a set is to convert the list into the set. The behavior symmetry of the list is broken into that of the set. The Adapter pattern captures this conversion process.

The Composite pattern breaks a homogeneous type hierarchy into a heterogeneous type hierarchy by allowing two types of object to cohabit: One type is the primitive, and another is the composite of the primitive. Although the Composite pattern presents the two types of object using a common interface, the behavior of the two types is different. One cannot substitute a primitive for a composite; one can only build a composite using the primitive.

The Decorator pattern breaks behavior symmetry by allowing individual objects to have unrelated behaviors. Decorator also breaks class symmetry, as it assigns responsibilities to individual objects, not to an entire class. The structure created by Decorator is a hierarchy of individual objects with different behaviors, rather than a type hierarchy, that has a uniform behavior.

Method symmetry is broken when overriding methods break the contravariant argument rule. Castagna [3] argues convincingly for breaking this rule and that covariance and contravariance characterize two completely distinct mechanisms in that the former is for code specialization and the latter is for subtyping. He proposes to capture both covariant and contravariant arguments in multi-methods. Yet since mainstream class-based languages are single-dispatch languages, Castagna's proposal is often only partially implemented. For example, Eiffel adopts the covariant policy on both argument and result types; Java requires that both argument and result types vary in unison, either in a covariant way or a contravariant way; C++ supports only covariant result type and does not care about argument type. Patterns, such as Override-Overload Method Pair (see www.curbralan.com), are introduced to simulate multi-methods in single-dispatch languages to support both covariant and contravariant method definitions.

There are more method symmetry breakers in [6] than there initially appears. For example, Template Method breaks method symmetry by separating the invariant part (template) of a method from its variant part (method steps). The same symmetry-breaking footprints can be found in Eiffel's Loop Invariants and other design patterns, such as Visitor, Interpreter, and Iterator [6]. Or should it be that all these method-breaker patterns are variations of Loop Invariants? It does not matter which answer is true or most appropriate; the heart of the matter is that method symmetry needs to be broken to allow flexible and reusable design. Other method breakers are Chain of Responsibility and Command [6], which break a method into sub-methods, again, to facilitate design flexibility.

Yet, while design patterns break symmetries enforced by original language features, they create their own local symmetries, and in so doing, they add extra features on top of languages.

The good news is, with judicious use, design patterns can provide some extra, useful programming utility; the bad news is, they can be so easily misused or, at the best, overused by pattern enthusiasts. The worst-case scenario is there are potentially an infinite number of design patterns resulting from the finite number of language features, all equally arbitrary; how do we know which ones are good and which ones are bad? Understanding the relationship between language fea-

tures and design patterns in terms of symmetry and symmetry breaking may provide the answer to that question.

CONCLUSION

In natural, as well as artificial systems, symmetry reigns supreme. This article has shown that class-based OO programming features create symmetries, whereas class-based design patterns break them. The relationship between language features and design patterns, as characterized by a more fundamental relationship between symmetries and broken symmetries, suggests a new way of understanding and viewing software design. The next time you are designing a program, think carefully about which symmetries you want to preserve and which ones you want to break, and why you want to do so. In a broad stroke, software design is fundamentally about making or breaking symmetry. **C**

REFERENCES

1. Abadi, M. and Cardelli, L. *A Theory of Objects*. Springer, NY, 1996.
2. Alexander, C. *The Nature of Order: An Essay on the Art of Building and the Nature of the Universe (Book One: The Phenomenon of Life)*. The Centre for Environmental Structure, 2004.
3. Castagna, G. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Language Systems* 17, 3 (1995), 431–447.
4. Coplien, J.O. C++ idioms. In N. Harrison, B. Foote, and H. Rohnert, Eds. *Pattern Languages of Program Design 4*. Addison-Wesley, Reading, MA, 2000.
5. Coplien, J.O. and Zhao, L. Symmetry breaking in software patterns. *Lecture Notes in Computer Science Series, LNCS 2177*, Springer, 2001.
6. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*. Addison-Wesley, Reading, MA, 1996.
7. Liskov, B. Data abstraction and hierarchy. *SIGPLAN Notices* 23, 5 (1988), 17–34.
8. Meyer, B. *Reusable Software*. Prentice Hall, 1994.
9. Potanin, A., Noble, J., Frean, M., and Biddle, R. Scale-free geometry in OO programs. *Commun. ACM* 48, 5 (May 2005).
10. Stewart, I. and Golubitsky, M. *Fearful Symmetry: Is God a Geometer?* Penguin, London, 1992.
11. Stroustrup, B. *The Design and Evolution of C++*. Addison-Wesley, 1994.
12. Weyl, H. *Symmetry*. Princeton University Press, 1952.

LIPING ZHAO (liping.zhao@manchester.ac.uk) is a senior lecturer in the School of Computer Science at the University of Manchester, U.K.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 2008 ACM 0001-0782/08/0300 \$5.00

DOI: 10.1145/1325555.1325564