

# Symmetry in Class and Type Hierarchy

Liping Zhao and James O. Coplien

Department of Computation  
University of Manchester Institute of Science and Technology  
PO Box 88, Sackville Street, Manchester M60 1QD, UK

Liping@co.umist.ac.uk; JOCoplien@cs.com

## Abstract

The class concept is central in OO programming (OOP) to implement abstract data types and enforce encapsulation. Type hierarchy has been regarded as a useful technique for consistency during extension. Yet, the importance of class and type hierarchy extends beyond their technical merits. Specifically, a class classifies objects and a type hierarchy classifies classes. Classification is fundamental to design, and its place in program evolution and programming language structure can be elucidated through formal symmetry models. Symmetry is the interplay between an invariant and the possibility of change. This paper discusses the importance of classification in OOP and shows the connection between classification and symmetry in class and type hierarchies.<sup>1</sup>

*Keywords:* Invariant, Eiffel, classification, group theory, object-oriented programming, Liskov substitutability principle.

## 1 Introduction

This article deals with three concepts: *symmetry*, *class*, and *type hierarchy*. Symmetry is the central concept in this article and requires a special introduction in Sections 2 and 3. Class and type hierarchy are familiar concepts so we give an overview here. At the opening of introducing object-oriented techniques, Meyer asked this question: What is the central concept of object technology (Meyer 1997, p. 165)? He continued:

Think twice before you answer “object”...

Objects remain important to describe the execution of an O-O system. But the basic notion, from which everything in object technology derives, is *class*...

Henderson-Seller also remarked:

In fact, “object-oriented” is really a misnomer because what we really should be talking about is “class-oriented,” since the essence of the object-oriented technique is actually the class (Henderson-Seller 1992, p. 34).

What is a class? Here are some authoritative answers:

- A class is an abstract data type equipped with a possible partial implementation. (Meyer 1997, p. 142)
- A class is a user-defined type. (Stroustrup 1997, p. 224)
- Data abstractions are supported by linguistic mechanisms in several languages... CLU provides a mechanism called a *cluster* for implementing an abstract type... In Smalltalk, data abstractions are implemented by *classes* (Liskov 1988).

Therefore a class<sup>2</sup> is a linguistic mechanism for implementing an abstract data type (ADT). Simply: a class implements an ADT; a class implements a user-defined type.

Why are classes so important to object technology? The answer, according to the standard OO literature (Liskov 1974, Liskov 1988, Meyer 1997), would cover the following points.

1. *A class implements an ADT.* ADTs are regarded as the theoretic basis for O-O programming (Guttag 1978, Liskov 1974, Liskov 1988, Meyer 1997). So the importance of a class implementing an ADT is self-evident.
2. *A class can be used to enforce encapsulation and information hiding.* Encapsulation and information hiding permit the representation of data to be changed locally without affecting programs that use the data. Locality is a key to modularity, which allows a program to be implemented, understood, or modified one module at a time (Liskov 1988).

In addition, Meyer added that a class is a module, a basic unit of software decomposition; a class can be used to enforce a uniform type system (Meyer 1997, pp.170-171).

Now, what is a type hierarchy? Liskov (1988) gives the following definition:

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a *subtype* is one whose objects provide all the behaviour of objects of another type (the *supertype*) plus something extra.

The property of a type hierarchy, also known as the Liskov substitution principle (LSP), defines what is required for a type hierarchy (Liskov 1988):

---

<sup>2</sup> The language constructs for implementing ADTs may not be necessarily called classes, but they achieve the same thing.

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 behavior of  $P$  is unchanged when  $o_1$  is substituted for  $o_2$ , then  $S$  is a subtype of  $T$ .

In other words, this property requires that a subtype must have the same operations of its supertypes and the operations must do the same things.

Type hierarchies can be implemented by the inheritance mechanism. However, it should be clear that a type hierarchy is not inheritance, but implemented by inheritance. Inheritance, as a language construct, can implement other hierarchies, such as implementation hierarchy and polymorphism (Liskov 1988).

However, besides its technical merits, a class has an important fact—perhaps a fundamental one—which remains implicit in OO literature: it *classifies* objects. Classification is essential in every science-related activity (Jensen 1986, Rosen 1989, Shubnikov 1974). Its existence in programming languages is not an accident and should not be overlooked. Perhaps the deeper meaning of classification is its connection to symmetry: Classification establishes the invariant relationship between a class and its members. Such a member-invariant class defines symmetry. Classification, at its heart, is symmetry (Rosen 1989).

When classes are arranged into a type hierarchy, it is another classification. Here a type hierarchy *classifies* classes. We believe that classes and type hierarchies are two basic organising structures in OO programming, upon which can other types of classification be established.

The purpose of this article is to bring out the importance of classification in OOP; it attempts to show that classification, when based on classes and type hierarchies, is symmetry. In doing so, we first give an introduction to basic symmetry concepts and formalism in Sections 2 and 3. We then show the link between symmetry and classification in Section 4 and discuss symmetry in classes and hierarchies in Sections 5 and 6. Section 7 looks at practical repercussions, applicability and ties to current and future work. We conclude the paper in Section 8.

## 2 Basic Concepts

Symmetry is one of the most fundamental concepts in science and art (Kappraff 1990, Ostdiek 1995, Pavlovic 1986, Rosen 1995, Senechal 1989, Shubnikov 1974, Urmantsev 1986). It is also an important and familiar notion in everyday life. However, in science and art symmetry has a precise mathematical definition. This section introduces some mathematical concepts of symmetry. These concepts establish the basis of a mathematical formalism of symmetry developed in Section 3.

### 2.1 Classic Symmetry and Group Definition

Classic symmetry is geometric: it is a *rigid motion*, a motion that leaves distances between points unchanged. These motions include *reflection*, *rotation*, *translation*,

and their various combinations (Senechal 1989, Shubnikov 1974, Weyl 1952). These motions are called symmetry operations, or transformations. The fundamental idea of these motions is *invariant change*. Generalised on this idea, the concept of symmetry extends beyond geometric motions, as Rosen (1989) observed:

Systems that might possess symmetry are not confined to the domain of concrete objects, but may be abstract to the extreme. The transformations involved do not have to be geometric; the imagination is free to roam: space, time, particle-antiparticle, permutation, and on to the abstract. Neither must the invariant aspect be appearance nor physical property, but may be any concrete or abstract aspect of the system under consideration. However, the very least we need for symmetry is the possibility of making a change and some aspect that is immune to this change.

Through the ages, symmetry remains to be a universal principle. To quote Rosen again: “I doubt whether there is a single field of scientific endeavour in which symmetry considerations cannot prove useful” (Rosen 1989, p. 193).

The theoretic study of symmetry is based on group theory. For every symmetry of a system, there is a corresponding symmetry group. We shall introduce symmetry groups in Section 3. Here, we give the mathematical definition of group.

A group is a set  $G$  together with a law of composition  $(a, b) \rightarrow ab : G \times G \rightarrow G$  satisfying the following four axioms:

1. *Closure*. For all  $a, b \in G$ , the set  $G$  is closed under composition:  $ab, ba \in G$ .
2. *Associativity*. For all  $a, b, c \in G$ , the composition is associative:  $(ab)c = a(bc)$ .
3. *Existence of an Identity Element*. For all  $a \in G$ , there exists an element  $e \in G$  such that  $ae = a = ea$ .
4. *Existence of Inverses*. For each  $a \in G$ , there exists an  $a^{-1} \in G$  such that  $aa^{-1} = e = a^{-1}a$ .

### 2.2 Mapping and Transformation

A *mapping* from set  $A$  to set  $B$  puts every element of set  $A$  in correspondence with some element of set  $B$ . The element of  $A$  that is in correspondence with an element of  $B$  is called an *object*<sup>Σ</sup> of the element of  $B$ . And the corresponding element of  $B$  is called the *image* of the element of  $A$ . A *transformation* is a mapping of an element from one set into the same set, such that every element has an image which is also an element. More than one element might have the same image; but not every element must necessarily serve as an image in the mapping. We denote a transformation  $T$  by

---

<sup>Σ</sup> Not to be confused with objects in O-O programs!

$$u \xrightarrow{T} v \quad \text{or} \quad T(u) = v$$

where  $u$  and  $v$  represent elements of a set,  $u$  is an object and  $v$  is an image of the object.

Transformations can be composed by consecutive application such that the composition of two transformations is defined as the result of applying one transformation to the result of the other, for example:

$$u \xrightarrow{T_1} v \xrightarrow{T_2} w \quad \text{or}$$

$$T_2 T_1(u) = T_2(T_1(u)) = T_2(v) = w$$

In general, transformations under the composition are not commutable. Among the transformations there are those that are *bijective* (one-to-one and onto), such that every element of a set serves as an image of the mapping and is the image of a unique object. Such bijective transformations are *invertible*; i.e., every bijective transformation has an inverse mapping obtained by reversing the transformation, denoted by  $T^{-1}$

$$v \xrightarrow{T^{-1}} u \quad \text{or} \quad T^{-1}(v) = u$$

An inverse mapping is also bijective so we say a bijective transformation and its inverse are mutual inverses. Bijectivity and invertibility imply one another. The compositions of mutual inverses in both orders give rise to the *identity transformation*  $I$ , such that:

$$\begin{aligned} T^{-1}T &= T T^{-1} = I & \text{or} \\ (T^{-1}T)(u) &= T^{-1}(T(u)) = T^{-1}(v) = u = I(u) & \text{or} \\ (T T^{-1})(v) &= T(T^{-1}(v)) = T(u) = v = I(v) \end{aligned}$$

The identity transformation is a mapping from an element onto the element itself, which can be seen as doing nothing or an empty transformation. It should be clear to the reader who is familiar with group theory that we are describing a *transformation group*. However, we shall delay our introduction to this concept until Section 3 when we describe symmetry transformation and symmetry group.

### 2.3 Equivalence Relation and Equivalence Class

Shubnikov and Koptsik (1974) offer the following definition of equivalence:

We shall call two objects equal in relation to some particular feature if both objects possess this feature.

For example, the vertices of a square are equal to one another in the sense that two edges at right angles meet in each of them. They are not, however, equal to one another with respect to their different orientations in space. All the sides of a scalene triangle are equal to one another in that each is a section of a straight line. The faces of a cardboard cube painted in different colours are equal to one another geometrically, but not with respect to colour.

The above examples show that *equivalence is a relative property*. Shubnikov and Koptsik (1974) stated: The concept of the *relative equality* of objects is of fundamental importance to the whole theory of symmetry. It allows us to specify the *measure of equality* or equivalence. In specifying a measure of equality for

certain aspect or feature, we may then say that the two objects (not just objects in OO programming!) are equal with respect to that aspect. This goes without saying—we shall say it anyway—absolute equality rarely exists in nature.

In mathematical terms, equivalence is defined as any *relation* (denoted by  $\equiv$ ) that might hold between the members of pairs of elements of a set and satisfies the following three conditions:

1. *Reflexivity*.  $a \equiv a$  for all  $a$  (every element of the set is equivalent to itself).
2. *Symmetry*.  $a \equiv b \Leftrightarrow b \equiv a$  for all  $a, b$ .
3. *Transitivity*.  $a \equiv b, b \equiv c \Rightarrow a \equiv c$  for all  $a, b, c$ .

For example, the arithmetic equality  $=$  is an equivalence relation. *Isomorphism* is another.

We can now introduce the concept of *equivalence class*<sup>II</sup>. Given a set of elements for which an equivalence relation is defined, any subset that contains a complete set of mutually equivalent elements is called an *equivalence class*. Thus an equivalence relation decomposes the set into equivalence classes, such that every element of the set is a member of one and only one equivalence class. Two different equivalence classes cannot have common elements.

## 3 Symmetry Formalism

This section gives a formalism of symmetry we developed for this paper. This symmetry formalism is based on a general symmetry formalism developed by Rosen (1995). We want to do so because symmetries we are looking for in programming languages and software are outside of the physical reality. Other generalised symmetry theories exist (Shubnikov 1974), but we find that Rosen's approach is closer to our understanding of symmetry and our perception of symmetry in programming languages and software.

### 3.1 Symmetry

Rosen offered a general definition of symmetry (Rosen 1995, p2):

**[Definition 1]** Symmetry is immunity to a possible change...

Note the two essential components of symmetry (Rosen 1995, p4):

1. *Possibility of a change*. It must be possible to perform a change on an object, although the change does not actually have to be performed.
2. *Immunity to the change*. Some aspect of the object would remain unchanged, if the change were performed.

<sup>II</sup> Not to be confused with classes in O-O programs!

In the above definition, a change is represented by a transformation. Immunity to the change means that the change brings the object into *coincidence with* itself. Simply, *symmetry means change, yet the same*.

We use the same example as Rosen: if we rotate an equilateral triangle around its center by  $120^\circ$ , it still looks the same (Figure 1). Here the rotation by  $120^\circ$  is a transformation or change. If we rotate it by  $240^\circ$ , it still looks the same. Rotate it by  $360^\circ$  and it returns back to its original position. Altogether an equilateral triangle has 6 symmetries: 3 rotations about the triangle's centre (by  $0^\circ$ ,  $120^\circ$ , and  $240^\circ$ ) and 3 reflections through three planes intersected with the triangle's three heights. The rotation by  $0^\circ$  is called the identity transformation of the triangle.

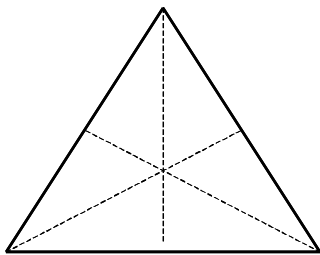


Figure 1. Equilateral Triangle (After Rosen (1995), p. 83)

As another example, a formula  $(A + B)$  remains invariant under the reflection through “+” with respect to its sum.

### 3.2 Symmetry Transformation

In Section 3.1, we stated that a change is represented by a transformation, but we deliberately kept the term transformation vague as our purpose was to make the two essential components of symmetry clear: change and invariant. We are now on the way to give a formal definition of symmetry:

**[Definition 2]** A symmetry of an object<sup>π</sup> is a bijective transformation that preserves the object invariance in the sense of the equivalence relation defined for the object.

This definition also contains the same two essential components as Rosen's definition in Section 3.1, but we use our own words because we want to make explicit some important meanings that are implicit in Rosen's definition:

- A symmetry transformation must be bijective or invertible (see the description of bijective transformation in Section 2.2.) Only then can it map an object onto itself without deformation.

For example, all the three rotations of the equilateral triangle discussed in Section 3.1 are invertible. The inverse of the rotation by  $0^\circ$  is itself, i.e., the identity transformation is its own inverse. The inverse of the rotation by  $120^\circ$  is the rotation by  $-120^\circ$ , etc.

Similarly, all the three reflections of the equilateral triangle are also invertible.

- A symmetry transformation preserves the object invariance with respect to the equivalence relation (see the description of equivalence relation in Section 2.3.) The invertibility of the transformation is the basic requirement for the invariance. However, *invariance is a relative property*, subject to the equivalence relation defined for it.

For example, when we say the equilateral triangle remains unchanged after the rotation by  $120^\circ$ , we actually mean that it *appears* unchanged or invariant, because we assume all the three vertices of the triangle are *equivalent*. Hence, immunity to the change implies an equivalence relation that we prescribe to the triangle. Such equivalence is relative; e.g., the vertices are not equivalent with respect to their orientations. Therefore, we say invariance is relative to the equivalence relation defined for it. *Given an equivalence relation, symmetry means change, yet the same*.

In Section 2.3, we stated that an equivalence relation for a set brings about a decomposition of the set into equivalence classes. We can rephrase the above definition of symmetry as follows.

**[Definition 3]** A symmetry of an object is a bijective transformation that preserves the invariance of the equivalence class membership of the object.

Consider the same equilateral triangle example. The triangle is invariant under any of those 6 symmetry transformations. Another way to say it is that the triangle has 6 *equivalent states* under these transformations. And these 6 equivalent states belong to the same equivalence class. Thus, the symmetry of the equilateral triangle is the immunity of equivalence class membership to a possible transformation of the triangle states.

### 3.3 Symmetry Group

We offer the following definition:

**[Definition 4]** A symmetry group of an object is a set of all invertible symmetry transformations mapping the object onto itself in the sense of the equivalence relation.

or:

**[Definition 5]** A symmetry group of an object is a set of all invertible symmetry transformations that leave the equivalence class of the object invariant.

It should be clear that Definitions 4 and 5 are based on Definitions 2 and 3. It should be noted that a symmetry group is a subgroup of a transformation group (Rosen 1995).

For example, the symmetry group of the equilateral triangle in Figure 1 is a group of 11 symmetry transformations. It consists of the identity transformation (the rotation about the triangle's centre by  $0^\circ$ ), whose inverse is itself; two rotations by  $120^\circ$  and  $240^\circ$ , and their inverses; three reflections through three planes intersected

<sup>π</sup> Object is a general concept here, which can be concrete or abstract.

with the triangle's three heights, and their inverses. It is a finite-order group, order-6.

We give another example from Rosen (1995, p82): Consider a given amount of pure gas as a system or object. The system has the microstates: the positions and momenta of all the molecules. The set of all allowable positions and momenta for all the molecules forms a state space of the system. The gas system also has macrostates, such as pressure, volume, and temperature. And all the pressures and temperatures may form another state space. The symmetry groups of this gas system are:

1. If microstates corresponding to the same macrostates are considered equivalent, the symmetry group is an infinite-order group, consisting of all the transformations that map microstates into equivalent macrostates only.
2. If macrostates of the gas with the same temperature are considered equivalent, for arbitrary invertible positive functions  $f(p, \dots)$  whose range of values is all positive numbers, the symmetry group is an infinite-order group of transformations of the form:

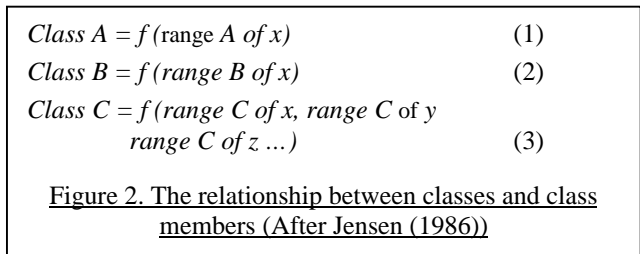
$$p \rightarrow p' = f(p, \dots)$$

The gas example shows that for the same object or system, there might be more than one symmetry group, depending on the aspects of the system under consideration and the equivalence relation defined for them.

#### 4 Classification as Symmetry

The importance of classification is evident in every science. Classification is a static, descriptive concept that establishes a many-to-one mapping from members to classes. Jensen (1986) expressed this many-to-one classificatory relation with the formula of Figure 2.

Classification is pervasive and important in biology and chemistry (Jensen 1986). For example, in chemistry, the establishment of the class *alkali metals* automatically implies the existence of member-invariant class statements, such that M is an air and moisture sensitive metal forming chlorides of the type MCl and sulphides of the type M<sub>2</sub>S.



Rosen states that the essence of classification is *analogy* (Rosen 1995, p.167):

A classification defines the analogy of belonging to the same class. If any set is decomposed into mutually exclusive classes, then the very property of belonging to the same class will define an analogy among the elements of the set.

And he pointed out that analogy is *symmetry* (Rosen 1995, pp. 164-167):

Analogy is the immunity of the validity of a relation or statement to changes of the elements involved in it.

For example, look at this analogy: An animal has a relatively long tail. Rosen stated that this statement makes all relatively long-tailed animals analogous in that, whatever their differences, they all possess the common property of having a relatively long tail. For example, Squirrel X has a relatively long tail; Squirrel Y has a relatively long tail; all the squirrels, whatever their differences in colour and race, are analogous. If we put all the squirrels in Class A in Figure 2, we can say Class A is valid for all x, where A represents the long-tailed animal statement and x represents squirrels. To show that an analogy is symmetry, we can check the long-tailed animal analogy with the two components of symmetry in Section 3:

- *Possibility of a change.* The analogy can be applied to more than one animal; the animal to which it is applied can be switched (from squirrel X to squirrel Y, to tiger Z).
- *Immunity to the change.* The analogy is valid to X, Y, Z, and just well to all the long-tailed animals.

What we try to converge here is that an analogy of belonging to the same class *implies* an equivalence relation between the members of the class. For example, all the animals that are analogous with respect to their relatively long tail are equivalent. A *classification thus defines an equivalence class*. Within such a class, all the long-tailed animals can be switched about and yet what makes the class true is still true. Such switching changes are symmetry transformations; they are invertible – switching squirrels A to B or switching B to A will leave the analogy invariant. If we don't perform any switch, it is the identity transformation. All these invertible transformations form a symmetry group, an infinite-order group. Therefore, classification is symmetry. Or more strongly, symmetry provides a basis for deriving structural classifications, as Shubnikov and Koptsik put forward (1974, pp. 309-310):

In crystallography and crystal chemistry, symmetry provides a basis for deriving descriptive morphological and structural classifications, grouping crystals in genetic series or in classes by appropriate sets of criteria. In exactly the same way, physics classifies elementary particles, atomic and molecular spectra, normal vibrations, etc. These symmetry classifications are based on various groups of permissible transformations carried out on the elements of the corresponding structures. The problem classification is a primary one for every science, so that symmetry, which establishes structural invariants, constitutes an essential technique for all of them.

What is all this to do with classes and type hierarchies in OO programs?

#### 5 Symmetry in Classes

What is a class? In Section 1 we cited from the literature that a class is an abstract data type. But then what is the

structure of a class, what is it made up of? Since a class is an ADT, one reasonable answer might be it is a collection of data members<sup>1</sup> and member functions (Guttag 1978). Elaborating on that, *we define the structure of a class as consisting of data elements and functional relationships between these elements*. This line of thinking, that the structure represents the fundamental elements of a phenomenon, their articulation, and their interactions, is consistent with that of the structuralists (Capra 1997, Glucksmann 1974). For example, in Figure 3, the structure of a *Time* class consists of the data members *hour*, *minute*, and *second*, and the functional relationships between them that set a time and print the time. In the figure, we have omitted the function definitions for simplicity.

```

class Time {
public:
    Time ( );
    void setTime ( int, int, int );
    void print24HourFormat ( );
    void print12HourFormat ( );
private:
    int hour;
    int minute;
    int second;
};

```

Figure 3. The Class Structure of Time

Given the *Time* class, we can then build time objects  $t_1, t_2, \dots, t_n$ . These objects are associated with one another by its class as expressed in Figure 2. These objects are equivalent in the sense that they belong to the same class *Time*. Belonging to the same class is thus the defining property of the equivalence relation for these objects. And this equivalence relation is characterised by the structure of the class. A class establishes the equivalence relation for a set of objects and makes these objects analogous with respect to their structure. The establishment of a class is a classification. A class, by a happy coincidence, is itself an equivalence class whose elements are objects. Following the same line of reasoning as in Section 4, we can show that a class defines a symmetry for its objects:

1. *Possibility of a change*. The class *Time* can be applied to more than one object; the objects to which it is applied can be mapped from one another (from  $t_1$  to  $t_2$ , to  $t_3$ ).
2. *Immunity to the change*. The class is valid to all these objects.

Alternatively, we can use the permutation group as an analogy to the class:

1. *Possibility of a change*. It is possible to rearrange the order of the objects within a class (from  $t_1, t_2, \dots, t_n$  to  $t_n, t_{n-1}, \dots, t_1$ ).
2. *Immunity to the change*. The class is immune to these permutations.

What is the symmetry group of this symmetry? What are the symmetry transformations in this group?

If we consider all the possible operations that can be applied to objects of the same class, we can perhaps say the only operations that can leave the class invariant are copying objects from one to another, mapping objects from one to another, or permutations of objects. Among these operations object copy has its linguistic realisation, whereas the other two are the imaginary mathematical operations. Such operations are essentially the same, as they switch about the objects in the same class. So mathematically, they can all be represented as a transformation  $S$  such that:

$$u \xrightarrow{S} v \equiv u \quad \text{or} \quad S(u) = v \equiv u$$

for all objects  $u$  in a class (Rosen 1995, p. 80) and by the equivalence relation such a transformation is invertible:

$$v \xrightarrow{S^{-1}} u \equiv v \quad \text{or} \quad S^{-1}(v) = u \equiv v$$

for all objects  $v$  in the class (Rosen 1995, p. 80). Thus  $S$  is a symmetry transformation.

We now prove that the set of all invertible symmetry transformations of the class for the equivalent relation forms a symmetry group.

1. Closure follows from transitivity of the equivalence relation. For symmetry transformations  $S_1$  and  $S_2$  on all objects  $u, v$ , and  $w$ , the composition of  $S_2S_1$  is the result of first applying  $S_1$  and then  $S_2$ , such that:

$$\begin{aligned}
 u \equiv v &= S_1(u), \\
 v \equiv w &= S_2(v) = S_2(S_1(u)) = (S_2 S_1)(u), \\
 u \equiv w &= (S_2 S_1)(u)
 \end{aligned}$$

Thus for all symmetry transformations  $S_1$  and  $S_2$  their composition  $S_2S_1$  is also a symmetry transformation, and by similar reasoning so is  $S_1S_2$ .

2. Associativity holds for composition by consecutive application. For all symmetry transformations  $S_1, S_2$ , and  $S_3$  on any object  $u$ , we want to prove:

$$S_3(S_2 S_1) = (S_3 S_2)S_1$$

Using the definition of composition in 1, we obtain from the left-hand side:

$$(S_3(S_2 S_1))(u) = S_3((S_2 S_1)(u)) = S_3(S_2(S_1(u))).$$

The right hand side gives:

$$((S_3 S_2)S_1)(u) = (S_3 S_2)(S_1(u)) = S_3(S_2(S_1(u))).$$

Hence we have proven the associativity.

3. The identity transformation is a symmetry transformation:

$$u \xrightarrow{I} u \equiv u \quad \text{or} \quad I(u) = u \dots u$$

for all objects  $u$  in the class.

4. The inverses exist because all the symmetry transformations are invertible, by definition.

<sup>1</sup> Data members can be defined by other classes, but for now, we don't want to know what the data members are made up of.

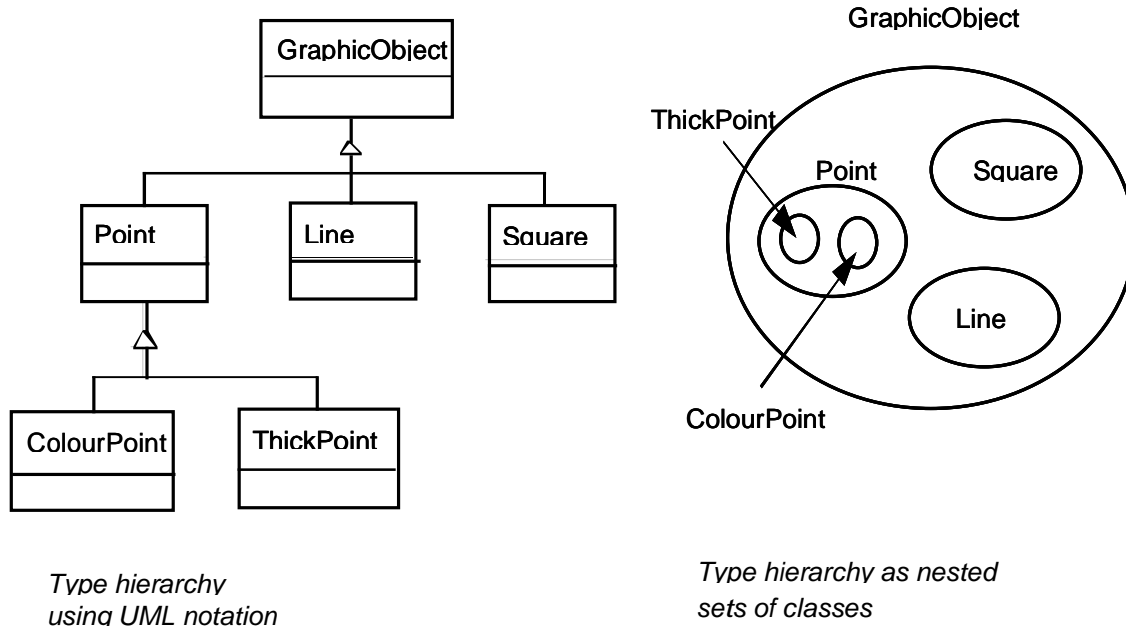


Figure 4. A Type Hierarchy – Subtyping Relationships between Classes

Therefore, all the invertible symmetry transformations defined for a class form a symmetry group for the equivalence relation. It is an infinite-order symmetry group, a subgroup of the infinite transformation group.

Hence we have proved that symmetry exists in classes and symmetry in a class corresponds to a symmetry group. This proof is to show that classes are of fundamental importance to OO programming in that *they define an invariant aspect of a system and preserve the structural integrity of the objects*. Therefore, class and class invariance are intimately related. *Underlying the class invariance is symmetry*. It should be mentioned at this point that the idea of *class invariant* in Eiffel (Meyer 1997) is in line with the symmetry considerations of classes discussed in this section, though it was driven by the pragmatic considerations of data values of the objects.

## 6 Symmetry in Type Hierarchies

The definition of type hierarchy has been given in Section 1. The process of defining a subtype from a supertype is called subtyping (Stroustrup 1997). To show symmetry in a type hierarchy, let us first look at an example. Figure 4 depicts a type hierarchy for a set of graphic object classes, such as *Point*, *Line*, *Square*, etc. It shows this type hierarchy in both UML notation and set notation. In this type hierarchy there are four subtyping paths from the *GraphicObject* type to its subtypes. In other words, substitutions can only take place within a single path, or when a subtype belongs to a subset of the supertype. For example, *ThickPoint* is a subset of *Point* which is a subset of *GraphicObject*. The set notation clearly shows why substitutions cannot cross the different paths.

Given a subtyping path in a type hierarchy, we say classes belonging to this path are equivalent in the sense of the LSP. For example, in the path *GraphicObject*  $\not\sim$

*Point*  $\not\sim$  *ColourPoint*, classes *GraphicObject*, *Point*, and *ColourPoint* are equivalent. Subtyping thus establishes the equivalence relation that brings about a decomposition of a type hierarchy into equivalent classes along each subtyping path. All the classes in a single subtyping path form an equivalence class, whose elements are classes, not objects. A subtyping path thus is a classification, which is an analogy, which is a symmetry. Here, we shall refrain ourselves from going through a detailed analysis and proof of this symmetry. The interesting reader can find the analysis and proof in (Coplien and Zhao 2000).

We have just shown that subtyping is about classification, which establishes the invariant relationship between a subtyping path and the classes belonging to this path. Therefore, a subtyping path establishes symmetry for its classes. Such symmetry conserves the type invariance. The type invariance is the ability to switch about objects between a supertype and its subtype. *The LSP defines the type invariance and symmetry permits it*. Recall that in Section 5, a type is a class that consists of data elements and functional relationships between these elements. Therefore, by preserving the type, the class is also preserved. And this is exactly what the LSP enforces—to make all the classes in a single subtyping path analogous with respect to their structure; all the classes in the path are the same, whatever their differences. We begin to see symmetry considerations and great programming ideas are converging. That symmetry is of fundamental importance to the design of software is reflected in the following writing (Coplien 1999, p. 65):

Variability makes sense only in a given commonality frame of reference. When we analyze variability, it is important to keep *something* fixed. Commonality analysis serves two purposes: to find the major dimensions of structure in a system and to provide a backdrop for variability analysis.

Indeed the idea of commonality and variability analysis is about the interplay of symmetry and asymmetry. After all, recognition of the similarities between the many patterns of structure makes possible a deeper appreciation of their differences.

## 7 Other Design Features and Applicability

Symmetry is at the foundation of many features of software design and programming. For example, the ties between symmetry and class invariants like those in Eiffel are straightforward and direct. Symmetry principles can be used as a generalised formalism to assess the “correctness” of a subtyping or inheritance lattice. The formalisms are stronger than the more commonly used intuitive notions of substitutability or commonality that appear in software design methods.

But beyond object-oriented programming, symmetry is a formalism that unifies many design approaches including object orientation, the commonality and variability analysis of domain engineering, and design patterns. Most programming language features can be viewed as symmetries (Coplien and Zhao 2000). While the main purpose of this paper has been to establish the links between symmetry and design formalisms, we can point to broader repercussions from related work (such as Coplien and Zhao 2000) and other work in progress.

Symmetry can provide a unifying formalism to deal with many of the problems of domain engineering. The multi-paradigm design approach published earlier by Coplien (1999) lends itself to direct use of the symmetry formalisms, where commonalities are the invariants of the symmetry and variabilities relate to the symmetry transformations. One fine point of multi-paradigm design is negative variability: a form of variation in software families that weakens the underlying commonality of the family. We propose this might be viewed as a form of symmetry breaking or as the introduction of local symmetries into the domain engineering formalisms.

Most central to our long-term interests is the place that symmetry plays in patterns. In physics a pattern is formally a configuration that results from irregularities in some physical process that introduces “symmetry breaking” or local symmetries in the physical structure of some system. Patterns likely can be treated as subgroups or semi-groups of the original system in which these phenomena or design considerations arise. As a physical example, consider the crown-shaped splash that arises from a drop of milk entering a large, shallow flat pan of milk: the original symmetry of the drop and of the milk surface becomes “broken up” by the process. Most processes in nature have an element of symmetry breaking. As a software example, the conflicting needs of inheritance and subtyping introduce a new local symmetry in object-oriented systems that use the Bridge pattern (Gamma et al. 1995) and many other patterns.

Symmetry provides a way of talking both about design patterns and about language features, and it is a potentially powerful tool for talking about the overlap between them. It might prove useful not only in language design but also in the analysis of programming language

intentionality. Vernacular programming languages like Turbo Pascal, C and C++ have features that express structure that breaks symmetry, or that at least violates commonality. C language unions or C++ template specialisations are good examples. Looking at these features from a symmetry perspective offers new insights not only on how these features interact with the “regular” (symmetric) features of the language, but they may offer insight into design itself. Nature—which is rife with asymmetry—may require such features of languages that are to remain viable in commercial application. Using the formalisms of symmetry and symmetry breaking, and their relationship to pattern theory, might yield new insights into language and design.

Last, we feel that symmetry provides a lingua franca for interdisciplinary dialogue between several fields of design and of the broader sciences. Physics, mathematics and biology already have deep roots in this theory, and the concepts and formalisms are familiar to many of the *beaux-arts*. With this broad foundation, the fledgling field of software design may be able to borrow insights from other fields that are centuries or millennia old.

## 8 Conclusion

The class concept plays an important role in OOP and its importance has been recognized as its ability to implement ADTs and enforce encapsulation. Type hierarchy has been regarded as a useful technique to enforce the type consistency in inheritance. Yet, the importance of class and type hierarchy extends beyond their technical merits. This paper shows why. Specifically, the paper attempts to make the following points:

- A class establishes the class invariance for its objects and a type hierarchy establishes the type invariance for its classes in a single subtyping path.
- A fundamental role of class and type hierarchy is classification, which distinguishes equivalent parts within a whole, while maintaining the essential generality of the structure and preserving invariant aspects of the system under consideration.
- Symmetry provides a basis for deriving structural classifications by deciding what is not going to be changed.
- Classification is a fundamental principle of organization and design, and it has more than incidental relationship to programming language structures.

## 9 Acknowledgements

Ralph Johnson provided many valuable comments and suggestions on our work. He is our sounding board in many respects. We gratefully acknowledge his help. We also appreciate the comments from the anonymous TOOLS Pacific 2002 reviewers and from Kevlin Henney.

## References

- CAPRA, F. (1997): *The Web of Life*. New York: Doubleday.
- COPLIEN, J. O. (1999): *Multi-Paradigm Design for C++*. Reading, MA: Addison Wesley.
- COPLIEN, J. O., and ZHAO, L. (2000): *Symmetry and Symmetry Breaking in Software Patterns*. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering (GCSE '2000)*. Erfurt, Germany.
- GAMMA, E., et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- GUTTAG, J.V. (1978): *The Algebraic Specification of Abstract Data Types*. *Acta Informatica* 10, 27-52.
- GLUCKSMANN, M. (1974): *Structuralist Analysis in Contemporary Social Thought*. London: Routledge & Kegan Paul.
- HENDERSON-SELLER, B. (1992): *A Book of Object-Oriented Knowledge*. Englewood Cliffs, NJ: Prentice Hall.
- JENSEN, W.B. (1986): *Classification, symmetry and the periodic table*. In *Symmetry: Unifying Human Understanding*. I. Hargittai (ed.). Oxford: Pergamon Press.
- KAPPRAFF, J. (1990): *Connections: The Geometric Bridge Between Art and Science*. New York: McGraw-Hill.
- LISKOV, B. and ZILES, S. (1974): *Programming with Abstract Data Types*. *SIGPLAN Notices*. 9(4), 50-59.
- LISKOV, B. (1988): *Data Abstraction and Hierarchy*. *SIGPLAN Notices*. 23(5), 17-34.
- MEYER, B. (1997): *Object-Oriented Software Construction*. 2<sup>nd</sup> Ed. Englewood Cliffs, New Jersey: Prentice Hall.
- OSTDIEK, V.J. and BORD, D.J. (1995): *Inquiry into Physics*. 3<sup>rd</sup> Ed. West Publishing Company.
- PAVOLOVIC, B. and TRINAJSTIC, N. (1986): *On symmetry and asymmetry in literature*. In *Symmetry: Unifying Human Understanding*. I. Hargittai (ed.), Oxford: Pergamon Press.
- ROSEN, J. (1995): *Symmetry in Science*. New York: Springer-Verlag.
- ROSEN, J. (1989): *Symmetry at foundations of science*. In *Symmetry 2: Unifying Human Understanding*. I. Hargittai (ed.). Oxford: Pergamon Press.
- SENECHAL, M. (1989): *Symmetry revisited*. In *Symmetry 2: Unifying Human Understanding*. I. Hargittai (ed.). Oxford: Pergamon Press.
- SHUBNIKOV, A.V. and KOPTSIK, V.A. (1974): *Symmetry in Science and Art*. Translated from Russian by G.D. Archard. New York: Plenum Press.
- STROUSTRUP, B. *The C++ Programming Language*. 3<sup>rd</sup> Ed. Reading, MA: Addison Wesley, 1997.
- URMANTSEV, Y. (1986): *Symmetry of System and System of Symmetry*. In *Symmetry: Unifying Human Understanding*. I. Hargittai (ed.). Oxford: Pergamon Press.
- WEYL, H. (1952): *Symmetry*. Princeton, NJ: Princeton University Press.