The Family Pattern*

Friedrich Steimann
KBS, Universität Hannover
Lange Laube 3, 30159 Hannover, Germany
steimann@acm.org

Abstract: Families are groups of classes that are related by *genus*, that is, by having one common ancestor. Typical examples of families are numbers, collections and geometrical shapes, which are usually realized as subhierarchies of a global class hierarchy. The Family pattern adds virtual instance creation to these subhierarchies and suggests that certain operations on family members should be implemented as factories returning instances of the appropriate type. Thus, the pattern ensures that clients of the family are always provided with instances that best suit their needs.

When studying the current literature on object-oriented design one cannot help but get the impression that class hierarchies, deep ones especially, have seriously fallen into disrepute.[1] This is a little surprising because since the time of Aristotle class hierarchies have served to structure our view of the world, and they have served their purpose well.[2] The recent resurgence of interest in ontology in the general context of modeling[3] underlines the apparent indispensability of type subsumption hierarchies in all knowledge structuring efforts.

Object-oriented design deals with creating software models of the real world; therefore, class hierarchies should have their rightful place in such models. The Family pattern describes a use of class hierarchies that is conceptually clean and practically foolproof, independent of the depth of the hierarchy and the number of classes involved.

Motivation

Certain types are naturally abstract, i.e., there are no instances of these types that are not instances of one of their concrete subtypes. For example, the instances of type `Number` must be instances of `Real`, `Integer`, or of any other concrete subtype of `Number`. Together, these types form families in the sense that they all are related by a common ancestor, the *root of the family* or *paterfamilias*, which specifies the properties shared by all its descendants. It is a natural, but not necessary, condition that only the leaf classes of families have instances; in biology, for example, there is no mammal that is not an individual of one concrete species. Likewise, every instance of class `Party` must either be a `Person` or an `Organization`.[4]

Usually, every type of the family has unique properties that distinguish it from its relatives. (In fact, the differences are the only reasons to specify subtypes.) And yet, there are many situations in which a client need not see nor care about (unless it desires to) the distinction between the different types — it simply requires that the instance it uses behaves as expected, that is, conforms to the interface specified by the abstract type, the root of the family. Such situations arise

- when the client deliberately wants to remain unspecific about the concrete type of the instance it resorts to (e.g., if it wants to speak of a party without saying whether it is actually a person or an organization); or
- if the instance is to undergo a series of transformations the outcome of which, although guaranteed to fall in the same family, can be of varying type.

Note that the former does not require that the root of a family such as `Party` should have instances of its own; rather, a person as well as an organization will do wherever a party is required (principle of substitutability). Examples of the latter occur when using a hierarchy of geometrical shapes (consisting of rectangles, squares, circles etc.) that undergo operations

* to appear in: *Journal of Object-Oriented Programming* (2001).

such as stretching and shearing; numbers (complex, real, rational, integer and natural) combined by the usual arithmetic operators; and collections (set, bag, sequence, ordered sequence etc.) which can be sorted, concatenated, and so forth.

An example

Numeric constraint satisfaction problems (NCSPs) involve variables the exact values of which are unknown, but are constrained by conditions (that, together with the variables, constitute the NCSP). Rather than being bound to a concrete value, a variable of a NCSP is therefore associated with a *set of possible values* (called its *range restriction*), which is usually a subset of the set of reals. Typically, these subsets are intervals, but attempts to solve a NCSP through repeated application of interval-manipulating operations may lead to restrictions that are non-contiguous[5]. An analysis of the problem leads to a class diagram like the one shown in Figure 1; the hierarchy rooted in the `SubsetOfReal` class is the `SubsetOfReal` family, and class `Variable` is a client of this family.

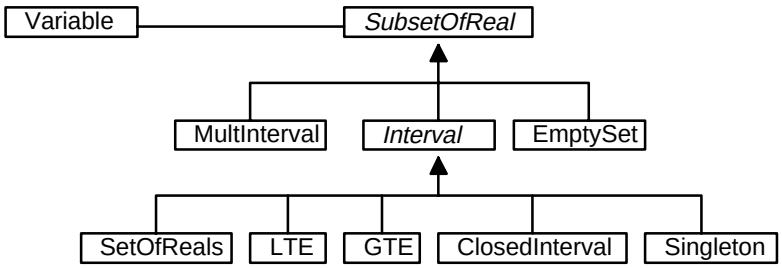

Figure 1: Class diagram of the `SubsetOfReal` family and its use by a client (in UML). `LTE` means *l*ess *t*han or *e*qual, i.e. the interval form negative infinity to an upper bound; `GTE` is defined accordingly.

Instance creation. Suppose the range of a variable must be restricted to the interval $[-1, 1]$. Such an interval is obtained by a call to a suitable constructor,

```
SubsetOfReal.interval(-1, 1)
```

returning an instance of class `ClosedInterval`. The call

```
Interval.interval(-1, 1)
```

should have the same effect since all `SubsetOfReal.interval` would do is call `Interval.interval` and return its result (Figure 2). The decisive difference is that in the former case, the client need not know about the existence of class `Interval` (or `ClosedInterval`, for that matter) — the principle of substitutability entails that all subclasses can remain hidden behind `SubsetOfReal` as long as it declares all the required operations. Note that constructors are virtual[6] since their return type may vary. For example,

```
Interval.interval(NEGATIVE_INFINITY, 1)
```

would return an instance of class `LTE` (*l*ess *t*han or *e*qual), which would also be obtained by calling

```
LTE.lTE(1)
```

(Figure 2). `MultiInterval`, which is the catch-rest class of the family, does not have a public constructor of its own; its instances are created as a result of operations performed on other classes' instances.

```
public abstract class SubsetOfReal {
    ...
    public static Interval interval(double lower, double upper) {
        return Interval.interval(lower, upper);
    }
    ...
}

public abstract class Interval extends SubsetOfReal {
    ...
    public static Interval interval(double lower, double upper) {
        ...
        if (lower == Double.NEGATIVE_INFINITY)
            return LTE.lTE(upper);
        ...
        // else
        return ClosedInterval.closedInterval(lower, upper);
    }
    ...
}

public final class LTE extends Interval {
    ...
    public static LTE lTE(double upper) {
        return new LTE(upper);
    }
    ...
}

public final class ClosedInterval extends Interval {
    ...
    public static ClosedInterval closedInterval(double lower, double upper) {
        return new ClosedInterval(lower, upper);
    }
    ...
}
```

Figure 2: Creator methods for the `SubsetOfReal` family (in JAVA)

Operators as factories. In the course of the solution of a NCSP, the ranges of the variables are further restricted, usually by repeatedly intersecting them with other ranges. However, this may entail a change of the range's type. For example, intersecting the `LTE` $(-\infty, 1]$ with the `GTE` $[-1, \infty)$ results in the `ClosedInterval` $[-1, 1]$ and, when subsequently intersected with $[1, 2]$, in the `Singleton` $\{1\}$. This poses two conditions on the implementation. First, in object-oriented programming languages that do not allow dynamic reclassification (the migration of an instance from one class to another), the intersection method must return a new instance of the appropriate type which replaces the variable's original range. Second, this type cannot be predicted statically, because it is not only determined by the types of the operation's operands, but also by their values. Therefore, the implementation of intersection must be that of a factory.[7]

Table 1 shows the signatures of the overloaded methods (including possible return types) that are to be implemented for the union and intersection operations of the `SubsetOfReal` family. In a JAVA implementation, every cell corresponds to one signature, the return type being the least common supertype of the cell's entries. Note that the inherent regularities (both operators are commutative; the set of reals and the empty set are neutral with respect to intersection/union; etc.) allow a reduction in the (otherwise combinatorial) number of signatures to be declared.

Table 1: Overloading of union and intersection operators and their possible return types (R = `SetOfReals`, M = `MultiInterval`, L = `LTE`, G = `GTE`, C = `ClosedInterval`, S = `Singleton`, E = `EmptySet`).

union	E	S	C	G	L	M	R		
R	R	R	R	R	R	R	R	R	R
M	M	R, M	R, M, L, G, C	R, M, G	R, M, L	R, M, L, G	M, L, G, C, S, E	M	M
L	L	M, L	M, L	R, M	L	L	M, L, C, S, E	L	L
G	G	M, G	M, G	G	G	C, S, E	M, G, C, S, E	G	G
C	C	M, C	M, C	C, S, E	C, S, E	C, S, E	M, C, S, E	C	C
S	S	M, S	S, E	S, E	S, E	S, E	S, E	S	S
E	E	E	E	E	E	E	E	E	E
		E	S	C	G	L	M	R	intersection

For domains such as numbers the fact that operations leave the states of their operands unchanged and instead return other instances is natural, since all instances are pre-existing and have no state. Typical examples of this are the `Boolean` and `Number` families from the SMALLTALK class hierarchy, excerpts of which are listed in Figure 3.

```
Object subclass: #Boolean

Boolean methods

isBoolean
    ^true
```

```
Boolean subclass: #False

False methods

& aBoolean
    ^false

eqv: aBoolean
    ^aBoolean not

not
    ^true

| aBoolean
    ^aBoolean
```

```
Boolean subclass: #True

True methods

& aBoolean
    ^aBoolean

eqv: aBoolean
    ^aBoolean

not
    ^false

| aBoolean
    ^true
```

```
Number subclass: #Integer

Integer methods

/ aNumber
    | numerator denominator gcd |
    aNumber class == Float
        ifTrue: [^self asFloat / aNumber].
    numerator := self * aNumber denominator.
    (denominator := aNumber numerator) < 0
        ifTrue: [
            denominator := 0 - denominator.
            numerator := 0 - numerator].
    (gcd := numerator gcd: denominator) = denominator
        ifTrue: [^numerator // gcd]
        ifFalse: [
            ^Fraction
                numerator: numerator // gcd
                denominator: denominator // gcd]
```

Figure 3: Excerpts of the `Boolean` and `Number` families from the SMALLTALK[8] class hierarchy. Note that the result of division defined in class `Integer` can have three different types: `Float`, `Integer`, and `Fraction`.

In other domains, however, operations are typically envoked to *alter the state* of an instance while *keeping its identity*. For example, the stretching of a rectangle that is a design element in a CAD document should retain the rectangle's identity, even if it so becomes a square. Implementing the different shapes as a `GeometricalShape` family (with `Rectangle` and `Square` as subclasses) seems natural, but leads directly to the instance migration (also called transmutation[1]) problem. Here, an additional class, `DesignElement`, can be introduced that mirrors the interface of `SubsetOfReal`. Each instance of this class holds an instance of `GeometricalShape` and delegates all geometrical transformations to this instance.[9] The CAD document can then be made up of design elements rather than geometrical shapes.

Where is the pattern?

The Family pattern is a complete subhierarchy (a class and all its descendants) that, for most cases, remains hidden behind the abstraction provided by the root of the hierarchy, the *paterfamilias*. The root declares all family-typical operations including instance creation for all concrete subclasses — it specifies the most general interface to the family that should be sufficient for all ordinary uses. More intimate knowledge of the subtree is necessary only when more specialized services are needed which are introduced by some subdivision of the family. In all

other cases, the client can rely on the *paterfamilias* to pick the member that serves its purpose best.

Instance creation is provided by virtual constructors defined with the root, and the concrete type of the created instance is determined by the constructor and the parameters of construction. All operations on the family's instances specified in the interface of the root are closed in the sense that the result is guaranteed to be a member of the family; yet it can be of varying type. Therefore, the family pattern frequently co-occurs with the envelope/letter idiom[6] and delegation[1].

A family can comprise subfamilies. For example, the `SubsetOfReal` family of Figure 1 subsumes the `Interval` family. A client needing an interval (and not any other type of subset) can directly turn to class `Interval`. However, `Interval` is not closed under the union and intersection operations, so that in the given example it is only of limited use.

Families offer different implementations of the same abstract data type. Therefore, the pattern is most useful when subtypes differ for representational or computational, not conceptual issues. Examples in this vein are the selection of a user interface depending on the number of data items to be presented, and the choice of the Fourier transform best suited to process the given data.[10] Yet there are many uses of the pattern in which the concrete types of the family differ conceptually. For example, the `Collection` family contains ordered and unordered sequences, and the concatenation of two ordered sequences is again a sequence, but not necessarily an ordered one. Similarly, the association of two persons is an organization, as is that of a person and an organization and the merger of two organizations.

Discussion

Two points are worthy of mention. First, one may argue that a suitable representation of multi-intervals (which are the most general of all forms of subsets of reals) is sufficient to cover all concrete classes of the `SubsetOfReal` family, thereby reducing the number of classes from nine to one and the number of operator implementations (for union and intersection) accordingly. However, since this class would have to cover all, special and general cases, its implementation will most likely be clumsier and, on average, computationally more expensive. In fact, in an effort to reduce the drawbacks of a single class implementation one might even end up implementing Table 1 as a set of nested switch statements. The Family pattern, on the other hand, breaks down the complexity of the problem into manageable chunks and distributes the code over different classes, each providing its own, optimized data structure thus allowing the code to be further simplified. The implementation of the `Boolean` type in SMALLTALK (Figure 3) is an illuminating, albeit trivial, example of this.

The second point is rather subtle. Purists might argue that `SetOfReals` should not be a subclass of `SubsetOfReal`, since the set of reals is really the superset of every subset of the reals and subclasses should stand for subsets, not supersets, of what their superclasses stand for[11]. However, while the class `SubsetOfReal` represents the *set of all possible subsets* of the set of reals (including the set of reals itself), all other classes of the family represent subsets thereof. In particular, the classes `SetOfReals` and `EmptySet` represent singletons since they have only one element (instance), namely the set of reals and the empty set, respectively. Thus, `SetOfReals` must be a sublcass of `SubsetOfReal`.

[1] P Coad, M Mayfield *Java Design: Building Better Apps & Applets* 2nd Edition (Yourdon Press, 1999).
[2] JF Sowa: *Conceptual Structures: Information Processing in Mind an Machine* (Addison-Wesley, 1984).
[3] W Swartout, A Tate "Ontologies" special section in *IEEE Intelligent Systems* (January/February 1999).
[4] M Fowler *Analysis Patterns: Reusable Object Models* (Addison-Wesley, Menlo Park 1997).
[5] E Hyvönen: "Constraint reasoning based on interval arithmetic: the tolerance propagation approach" *Artificial Intelligence* 58 (1992) 71–112.
[6] JO Coplien *Advanced C++ Programming Styles and Idioms* (Addison-Wesley, Massachusetts 1992).
[7] F Steimann "Abstract class hierarchies, factories, and stable designs" *Communications of the ACM* (in press).
[8] *Smalltalk Express* (freely available for download on the Internet).
[9] the envelope/letter idiom as described by Coplien[6]; *cf.* also delegation in JAVA[1].
[10] JW Cooper: "Using design patterns" *Communications of the ACM* 41:6 (1998) 65–68.
[11] DM Papurt "Generalization and polymorphism" *Report on Object Analysis & Design* 2:5 (1996) 13–16.