The Good, the Bad, and the Koyaanisqatsi

Consideration of Some Patterns for Value Objects

Kevlin Henney
kevlin@curbralan.com
kevlin@acm.org

May 2004

Abstract

koyaanisqatsi */ko.yaa.nis.katsi/ (from the Hopi language), noun:*

1. *crazy life.*
2. *life in turmoil.*
3. *life disintegrating.*
4. *life out of balance.*
5. *a state of life that calls for another way of living.*

From the 1983 film of the same name, directed by Godfrey Reggio, soundtrack composed by Philip Glass

This paper considers and combines some seemingly disparate ideas: symmetry within and across class interfaces; the notion of ineffective but recurring design styles as patterns, albeit dysfunctional ones; the role of balance within good patterns; value-based programming idioms in Java. Inappropriate symmetry in a faulty but common interface style is presented as a plausible pattern, which is then counterbalanced by a pair of patterns that break and recast the symmetry at a different level, resolving the design tension.

Pairing every `get` method with a `set` method offers an apparently simple and symmetric design style, but it is one fraught with problems. In the context of value-based programming in Java the balance may be better expressed across two classes: an IMMUTABLE VALUE and a MUTABLE COMPANION. These complementary patterns represent two sides of the same design coin, a symmetry and contrast at a different level to the per-method pairing of `get` with `set`. In contexts other than value-based programming in Java the design tension will be resolved in different ways. This is not to say that one should never write `get` and `set` methods, just that to characterize the practice as the GETTERS AND SETTERS pattern suggests that the use of `get` and `set` methods is a good solution to a specific problem rather than a cumulative consequence of other design decisions. Reasoning more carefully about GETTERS AND SETTERS as a pattern, with documented forces and consequences, exposes its unbalanced nature and its weakness as a design guideline.

Introduction

Symmetry is a notion grounded in our experience of the real world. We apply it, by metric and by metaphor, to other domains, both physical and abstract. It can be considered both formally and informally, with respect to a rigorously mathematical, mirror-perfect balance or as a looser, more tacit equilibrium of opposites. Symmetry is a form of consistency that is about balance and opposites.

Symmetry can be a useful, memorable, and harmonic local property in designs [Alexander2002, Coplien+2001, Zhao+2003]. It requires less effort to explain and recall, guiding expectation that when a particular feature is present, its logical counterpart will also be there. That where there is the capability for output there is also the capability for input. That where a resource can be acquired it can also be released. That where there is a `commit` there is also a `rollback`.

It is tempting to extrapolate this series of balanced pairings to `get` and `set` methods, i.e. where there is a `get` method there should also be a `set` method. However, this would be a step too far and a temptation that should be resisted. Such a guideline is simplistic rather than simple. It leads to code that is difficult to use correctly and, except in trivial cases, difficult to implement correctly. In spite of this, some company coding guidelines mandate a pairing of a `set` with every `get`, and even a `get` for every private field. Others have gone even further by arranging for a `get` and a `set` to be generated automatically for every field. A simple vowel shift might better describe such code: *uncapsulated*.

This style is sometimes favorably referred to as a pattern. Patterns can be considered either "good" or "bad". When programmers normally talk about patterns there is an implicit assumption that they are talking about patterns that improve the quality of their systems, an implicit assumption that documented patterns are, almost by definition, of the "good" variety, so that anything that is a "bad" pattern is not a pattern. From this perspective, automatic pairing of `set` with `get` methods cannot be considered a pattern, only a questionable and smelly [Fowler1999] coding guideline with a long footnote of problems. *Caveat setter*.

Any pattern describes a dialog with a design situation. It explores the context of a design problem, whether large scale or fine grained, enumerating the forces that drive and buffet the design. A pattern moves on to describe a solution. But, importantly, a pattern does not end there. The dialog continues by describing the consequences of applying the proposed solution, detailing the resulting context. What forces were balanced? What was left unresolved? What benefits have arisen? What liabilities have been incurred?

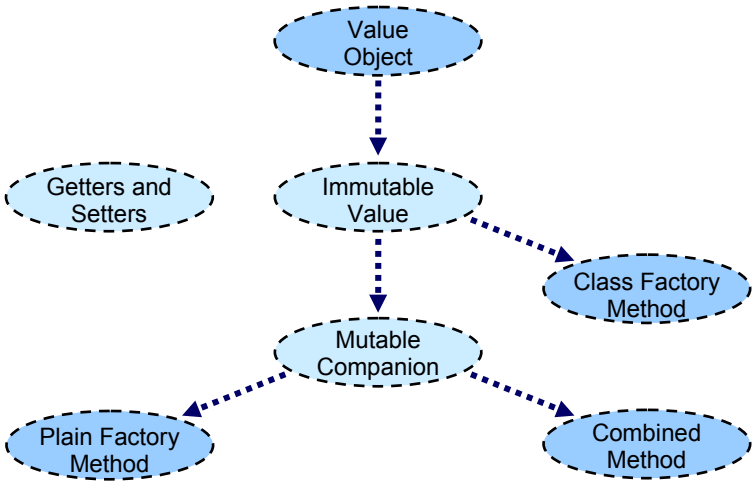
If "bad" patterns are also considered to be patterns — the case originally [Alexander1979] — we can understand their dysfunctionality by trying to consider each as an unfulfilled whole — a whole with holes. A "good" pattern is one whose forces are both genuine and well matched by its consequences. A "bad" pattern, on the other hand, is one whose forces are incomplete and whose consequences are out of balance.

Patterns, whether good or bad, have a recurring nature and recognisable form. By trying to understand GETTERS AND SETTERS as some kind of pattern we can see it more easily in code; we can better understand why it is not a good pattern; most importantly, we can see how we might better address the design problem — preferably with a solution rather than another problem. This is not to say that objects should never have paired `get` and `set` methods — if there is any guideline here, it is probably that any `set` methods should be accompanied by `get` methods, rather than the other way around [Henney2004] — just to point out that when useful, such a pairing is a consequence of other design decisions and less a decision in itself.

This paper takes the implementation of VALUE OBJECT types in Java [Fowler2003, Henney2000a, Henney2000b, Wiki] as its example for comparing "good" and "bad" patterns. GETTERS AND SETTERS can in principle apply to any kind of object in any language supporting objects, but the context of VALUE OBJECTs allows discussion of an alternative approach to be specific and concrete — in this case, the IMMUTABLE VALUE and MUTABLE COMPANION pairing. In another context, such as expressing the interface of an entity type or refactoring procedural code into object-oriented code, GETTERS AND SETTERS would be presented a little differently, taking into account differences in the context, forces, and consequences while retaining the similar recurring intention and construction that makes it some kind of pattern.

The complementary IMMUTABLE VALUE and MUTABLE COMPANION patterns are part of a larger pattern language, but are presented in this paper connected only to VALUE OBJECT, one another, and variants of FACTORY METHOD [Henney2003c]. GETTERS AND SETTERS is persuasive but deceptive: it is based on a slightly false identification of forces and a selective consideration of consequences. Closer inspection and reflection reveals that its liabilities outweigh its benefits. Although the pattern has an air of plausibility in its presentation, it is out of balance: any application of it can throw a design off centre. The complementary pattern pairing considers more authentic design criteria with forces that are relevant and an outcome that is more clearly balanced.

The following diagram shows the generative relationships (or lack of) between the patterns under discussion, highlighting the patterns described in this paper:



It is sometimes said that no pattern is an island, but, as the diagram shows, this is not always true. Although, in the documented form that follows, GETTERS AND SETTERS believes that it has VALUE OBJECT as its context, the relationship, from VALUE OBJECT's perspective, is an unconsummated one. A careful examination of GETTERS AND SETTERS' consequences indicates that what at first appears to be a beneficial pattern, is little more than a chimera. It has a false symmetry encouraged by assonance in its English form — a phony euphony. A *setter* does not in truth have the opposite effect to a *getter*. It is perhaps more informative and constructive to think in terms of *putter* or *modifier* and *enquirer* or *query*, evaluating the need for these roles specifically for each design. Such design symmetry as exists is local but not necessarily at the level of a single class interface: it can be found more convincingly broken and recast across two class roles, IMMUTABLE VALUE and MUTABLE COMPANION.

GETTERS AND SETTERS — A Dysfunctional Pattern

A VALUE OBJECT can be described in terms of one or more primitive values. A VALUE OBJECT provides a higher level means of describing information in the system than simply holding or passing around one or more primitive values. A VALUE OBJECT type describes a concept more precisely, including encapsulated operations that refer to the VALUE OBJECT as a whole. Each primitive value associated with a VALUE OBJECT is an attribute that has a specific role. However, exposing such attributes as public data is considered to be a poor practice.

Rather than communicate a concept, such as a date, in terms of primitive values, e.g. three `int` arguments, a VALUE OBJECT expresses the concept as a single object. This packaging is both easier to comprehend and simpler to manipulate. An object branded as a `Date` is clearly more meaningful than an ad hoc grouping of three otherwise unconstrained numbers.

However, if attributes, such as year, month, and day, are exposed as public fields, the strength of encapsulation is weakened:

```
public class Date
{
    public int year, month, day;
    ...
}
```

Constraint enforcement is lost: there is no 29th February 2003 and there is never a 32nd January. Public data advertises that the exposed fields have no relationships or rules governing them that the class author wishes to enforce.

Public data also commits the class to a single representation. Although it conveniently matches our intuition, representing a date as three integer values is not the most effective or efficient representation for most programs [Henney2003b]. A scalar epoch-based date, where the date is counted in days from a fixed date, is often simpler to work with and more compact in representation, e.g. a single integer counting the days since 1st January 1900. The year, month, and day attributes are still conceptually valid, but they would have to be calculated from the representation instead of actually being the representation. A similar case can be made for the day of the week and the week in the year: conceptually valid attributes that can be calculated, but not ones that should be stored and exposed publicly as fields along with other attributes.

Therefore, for each primitive value that can be considered an attribute of a VALUE OBJECT, provide a pair of methods that allow the attribute to be queried and set. An attribute will often correspond to a private field, but this need not be the case: an attribute may be a calculated value, derived from other fields.

The most common naming convention is to prefix the name of the conceptual attribute with a `get` and a `set` for each pair of methods:

```
public class Date
{
    public Date(int year, int month, int dayInMonth) ...
    public int getYear() ...
    public void setYear(int newYear) ...
    public int getMonth() ...
    public void setMonth(int newMonth) ...
    public int getDayInMonth() ...
```

```
    public void setDayInMonth(int newDayInMonth) ...
    ...
}
```

However, this is not necessarily the clearest or cleanest option. GETTERS AND SETTERS can be implemented using slightly less pedestrian names, e.g. `year` instead of `getYear`.

Different implementations with different trade-offs can be expressed using a common interface:

```
public class Date
{
    ...
    private int year, month, day;
}
```

Or:

```
public class Date
{
    ...
    private int daysSince1900;
}
```

A problem with the fine granularity of the operations is that of invalid, intermediate states. Consider the following:

```
Date date = new Date(2003, 2, 28);
date.setDayInMonth(30);
date.setMonth(1);
```

The intent is that the object referred to by `date` is initialized to 28th February 2003 and then modified to 30th January 2003. The problem is that the ordering of the operations shown means that at one point the object would conceptually have to hold the date 30th February 2003, an invalid date. Either the operation must fail at this point or the validity enforcement of the class must be disabled, weakening its encapsulation. For attributes that are derived rather than stored directly, this can become even more of a challenge: 30th February 2003 could be interpreted as 2nd March 2003, which would lead to the final date being calculated as 2nd January 2003.

Another liability arises from sharing. A `Date` object that is shared by being passed as an argument or returned as a method result can be modified by one party in ways unexpected and unwanted by the other party. The only solution in this context is careful and defensive copying of arguments and results, so that each party holds a unique instance.

IMMUTABLE VALUE — A Better Pattern

VALUE OBJECTs are fine-grained, stateful objects used to express quantities and other simple information in a system. Object identity is not significant for a value, but its state is. VALUE OBJECTs form the principal currency of representation and communication between many kinds of objects, such as entities and services. As such, references to VALUE OBJECTs are commonly passed around and stored in fields. However, state changes caused by one object to a value can have unexpected and unwanted side effects for any other object sharing the same value instance.

How can you share VALUE OBJECTs and guarantee no side-effect problems? Defensive copying is a convention for using a modifiable value object to minimize aliasing issues. However, this practice is tedious and error prone, and may lead to excessive creation of small objects, especially where values are frequently queried or passed around.

With multi-threading the troublesome consequences of aliasing are multiplied. Synchronizing methods addresses the question of valid individual modifications, but does nothing for the general problem of sharing. Synchronization also incurs a performance cost.

Therefore, define a VALUE OBJECT type whose instances are immutable. The internal state of a VALUE OBJECT is set at construction and no subsequent modifications are allowed: only query methods and constructors are provided; no modifier methods are defined. A change of value becomes a change of VALUE OBJECT referenced.

The absence of any possible state changes means that there is no reason to synchronize. Not only does this make IMMUTABLE VALUEs implicitly thread safe, but the absence of locking means that their use in threaded environments is also efficient. Sharing of IMMUTABLE VALUEs is also safe and transparent in other circumstances, so there is no need to copy an IMMUTABLE VALUE, and thus no need to support cloning or copy construction.

Declaring the fields `final` ensures that the *no change* promise is honoured. This guarantee implies also that either the class itself must be `final` or its subclasses must also be IMMUTABLE VALUEs:

```
public final class Date
{
    public Date(int year, int month, int dayInMonth) ...
    ... // other constructors
    public int year() ...
    public int month() ...
    public int dayInWeek() ...
    public int dayInMonth() ...
    public int dayInYear() ...
    ... // other query methods
    private final int daysSince1900;
}
```

References to — rather than the attributes of — an IMMUTABLE VALUE are changed to effect value change. The reference may be to an existing object or a new one may need to be created. There are complementary techniques for creating IMMUTABLE VALUEs: provide a complete and intuitive set of constructors; provide a number of CLASS FACTORY METHODs [Henney2003c] if some aspect of the creation needs to be controlled or named, e.g. `Date.today()`; provide a MUTABLE COMPANION if values are used in complex expressions that could generate many IMMUTABLE VALUE instances. Values are not resources, so their FACTORY METHODs do not need to be mirrored by DISPOSAL METHODs [Henney2003c].

MUTABLE COMPANION — A Complementary Pattern

IMMUTABLE VALUE objects provide a simple and safe means of expressing and sharing values in a system. However, the construction of an IMMUTABLE VALUE is not always a simple matter of using a `new` expression or calling a CLASS FACTORY METHOD. Some values may be the outcome of complex or ongoing calculations.

Constructors for an IMMUTABLE VALUE type offer a way of creating instances from a fixed set of arguments, but they cannot accumulate changes or handle complex expressions without themselves becoming too complex or overly general. The need for frequent or complex change typically leads to expressions that create many temporary objects.

For example, concatenating multiple strings, stripping unwanted characters and tokens out of an input line before further processing, accumulating an invoice total from many items, and generating a sequence of dates approximately two weeks apart but not falling on public holidays or weekends are all tasks that would involve the creation — and rapid neglect — of many IMMUTABLE VALUEs. What may, in isolation, be considered a relatively minor space and time overhead can hit a program's performance and memory recycling limits when frequent.

Therefore, provide a companion class for the IMMUTABLE VALUE type that supports modifier methods. A MUTABLE COMPANION instance acts as a factory for IMMUTABLE VALUE objects. For convenience this factory can stand not only as a separate class, but can also take on some of the roles and capabilities of the IMMUTABLE VALUE.

The modifier methods allow for cumulative or complex state changes. They should be `synchronized` COMBINED METHODs [Henney2000c] if shared use in a multi-threaded environment is intended, but un-`synchronized` otherwise. A PLAIN FACTORY METHOD [Henney2003c] allows users to get access to the resulting IMMUTABLE VALUE:

```
class DateManipulator
{
    ... // constructors, modifier, and other query methods
    public synchronized void set(int year, int month, int day) ...
    public synchronized Date toDate() ...
    ... // private representation
}
```

A MUTABLE COMPANION should not, however, have an inheritance relationship with its corresponding IMMUTABLE VALUE. A mutable object is not truly substitutable for a type whose usage contract is founded on its immutability. Such non-substitutable inheritance would reintroduce the sharing problems eliminated by using an IMMUTABLE VALUE. Because the role of a MUTABLE COMPANION is to create IMMUTABLE VALUEs, rather than to be used as a value in its own right, neither cloning nor copy construction is a requirement.

The core Java `String` and `StringBuffer` classes are canonical examples of an IMMUTABLE VALUE type with a MUTABLE COMPANION. With the exception of a transparent caching optimization for its hash code, a `String` instance is immutable and freely shareable across threads. A `StringBuffer` has a `synchronized` method interface with the standard object-as-string query, `toString`, doubling as the PLAIN FACTORY METHOD. Both classes are `final` and neither has any familial relationship with any class but `Object`.

A MUTABLE COMPANION is not always necessary, and should be provided only when the cost or complexity of working with IMMUTABLE VALUEs alone becomes inappropriate. Appropriateness is not so much subjective as relative: it depends on both the type that the IMMUTABLE VALUE models and the specific application of that type.

Acknowledgments

This paper is derived from a previous article [Henney2003a].

I would like to thank Joel Jones for his shepherding of this paper for VikingPLoP 2003 and Neil Harrison for his comments following the conference. From the workshop at the conference I would like to thank Jacob Borella, Franco Guidi-Polanco, Alan O'Callaghan, and Titos Saridakis. For their later comments on the conference draft I would also like to thank Phil Hibbs and Hubert Matthews.

References

[Alexander1979] Christopher Alexander, *The Timeless Way of Building*, Oxford, 1979.

[Alexander2002] Christopher Alexander, *The Nature of Order, Book 1: The Phenomenon of Life*, Center for Environmental Structure, 2002.

[Coplien+2001] James Coplien and Liping Zhao, "Symmetry Breaking in Software Patterns", *Springer Lecture Notes in Computer Science*, October 2001, `http://www.bell-labs.com/user/cope/Patterns/Symmetry/Springer/SpringerSymmetry.html`.

[Fowler1999] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Fowler2003] Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[Henney2000a] Kevlin Henney, "Patterns of Value", *Java Report* 5(2), February 2000, available from `http://www.curbralan.com`.

[Henney2000b] Kevlin Henney, "Value Added", *Java Report* 5(4), April 2000, available from `http://www.curbralan.com`.

[Henney2000c] Kevlin Henney, "A Tale of Two Patterns", *Java Report* 5(12), December 2000, available from `http://www.curbralan.com`.

[Henney2003a] Kevlin Henney, "Unvollendete Symmetrie in Java", *JavaSPEKTRUM*, May 2003, available in English as "Unfinished Symmetry" from `http://www.curbralan.com`.

[Henney2003b] Kevlin Henney, "The Taxation of Representation", *The Road to Code* blog at *Artima*, 30th July 2003, `http://www.artima.com`.

[Henney2003c] Kevlin Henney, "Factory and Disposal Methods", *VikingPLoP 2003*.

[Henney2004] Kevlin Henney, "Opposites Attract", *Application Development Advisor*, to be published.

[Wiki] `http://c2.com/cgi/wiki?ValueObject`.

[Zhao+2003] Liping Zhao and James Coplien, "Understanding Symmetry in Object-Oriented Languages", *Journal of Object Technology* 2(5), September–October 2003, `http://www.jot.fm/issues/issue_2003_09/article3`.