Patterns in Java
Unfinished Symmetry

Kevlin Henney
kevlin@curbralan.com

Koyaanisqatsi[1]

Symmetry is about balance. The expectation that when a particular feature is present, its logical counterpart will also be there. Where there is the capability for output there is also the capability for input. Where a resource is acquired it can also be released [Henney2002a, Henney2002b]. 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 write correctly.

I have sometimes seen this style 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. From this perspective, automatic pairing of `get` and `set` methods cannot be considered a pattern, only a coding guideline with a long footnote of problems. *Caveat setter*.

Any pattern describes a dialogue with a design situation. It explores the context of a design problem — whether large scale or fine grained — and enumerates 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 dialogue 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 [Alexander1979], we can understand their problems by understanding their consequences. A good pattern is one whose forces are genuine and that are well matched by its consequences. A bad pattern, on the other hand, is one whose forces and consequences are out of balance. Once again, a question of symmetry.

Patterns, whether good or bad, have a recurring nature and recognisable form. By understanding Getters and Setters as a pattern, we can see it more easily in code, we can better understand why it is not a good one and, most importantly, we can see how we might better address the design problem.

To keep the discussion concrete, let's focus on implementing Value Object types in Java. [Fowler2003, Henney2000a, Henney2000b, Wiki]. Although Getters and Setters can in

---

[1] Taken from the 1983 film directed by Godfrey Reggio and soundtrack composed by Philip Glass, *Koyaanisqatsi* means "life out of balance" in the Hopi language.

principle apply to any kind of object in any language supporting objects, providing a reasonable and reasoned alternative requires a more specific and restricted problem and solution scope. Sometimes Getters and Setters on other object types may be resolved in a similar way; sometimes a quite different solution would be more appropriate, especially in other languages.

('Bad') Pattern: Getters and Setters

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:

```
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 also data 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. An 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 should not 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:

```
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 more reasonable names, e.g. `year` instead of `getYear`.

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

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

Or:

```
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 initialised 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 comes 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.

('Good') Pattern: Immutable Value

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 object types, 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.

In a multi-threaded environment the consequences of sharing and changes can be multiplied. Synchronising methods addresses the question of valid individual modifications, but does nothing for the general problem of sharing. Synchronisation of change 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 synchronise. Not only does this make Immutable Values implicitly thread safe; 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.

Declaring the fields `final` ensures 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.

If a value with different attributes is required, a new object is created or found with the desired value. Immutable Value references are changed rather than attributes. There are complementary techniques for creating Immutable Values: provide a complete and intuitive set of constructors; provide a number of Class Factory Methods; provide a Mutable Companion. Values are not resources, so their Factory Methods do not need to be mirrored with Disposal Methods.

('Good') Pattern: Mutable Companion

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.

Therefore, provide a companion class for the Immutable Value type that supports modifier methods. A Mutable Companion instances 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, which should be `synchronized` if use in a multi-threaded environment is anticipated, allow for cumulative or complex state changes. A Factory Method allows users to get access to the resulting Immutable Value:

```
class DateManipulator
{
    public synchronized Date toDate() {...}
    ...
}
```

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 based almost entirely on its immutability. Such non-substitutable inheritance would reintroduce the sharing problems eliminated by using an Immutable Value.

Conclusions

Symmetry is a useful property in designs. It means that features are balanced, making them easier to remember. They require ultimately less explanation than designs that do not have a simple symmetric model. For example, many methods become easier to comprehend and work with if they are provided are in balanced families.

However, symmetry is not the only answer. Some of the best design is transparent. For instance, fully encapsulated resource management does not require the involvement of an external object user. Where a transparent design is not possible a symmetric one offers the next simplest option. For instance, if the resource used should be parameterizeable, it must be visible in the interface of an object, so transparency is not an option but symmetry is.

And asymmetry does not always imply complexity. Asymmetry can be simpler when it can be defined in terms removal of a feature from the corresponding symmetric design, i.e. it is simpler by omission, with the omission often arising from a simplifying assumption. For example, resources are kinds of objects that are scarce, which drives the need for their management, and suggests a symmetric design that reflects an acquisition and release cycle. Unless the target system is memory constrained, value objects do not really represent a scarce resource, so only their creation needs to be addressed. In this case, *not* having Disposal Method [Henney2002a] is both simple and sufficient. However, many asymmetric designs do not fall into this simpler category. They are often more complicated than equivalent symmetric designs because they include more detail and appear less well ordered. Asymmetry is the exception rather than the rule, but it is not ruled against.

When defining value objects, Getters and Setters offers an apparently symmetric design, but one that is fraught with problems. The symmetry may be better expressed across two classes: an Immutable Value and a Mutable Companion [Henney2000a, Henney2000b]. These complementary patterns represent two sides of the same design coin. This is not to say that one should never write `get` and `set` methods; just that characterising the practice as a pattern suggests that the use of `get` and `set` methods is the solution to a specific problem rather than a cumulative consequence of other design decisions.

In terms of individual patterns there is also a symmetry to be observed. The forces and consequences should balance. Sometimes the perceived problem and forces for a pattern are not the ones that are truly relevant in a design, which can throw a design off centre.

References

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

[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`.

[Henney2002a] Kevlin Henney, "Symmetrie in Java", *JavaSPEKTRUM*, July 2002, available in English as "The Importance of Symmetry" from `http://www.curbralan.com`.

[Henney2002b] Kevlin Henney, "Die Gratwanderung zwischen Symmetrie und Asymmetrie", *JavaSPEKTRUM*, September 2002, available in English as "The Temptations of Symmetry" from `http://www.curbralan.com`.

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