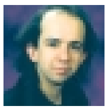

Symmetry has something to offer the world of software. Kevlin Henney stands some interfaces in front of the mirror to see what's missing

# OPPOSITES ATTRACT

TOGETHER, OPPOSITES CAN FORM A memorable whole – love and hate, yin and yang, elves and dwarves, Fred Durst and Britney Spears. The notion of such symmetry guides expectations[1, 2]. Sometimes the absence of symmetry is jarring and inconvenient; its presence simplifying and predictable.

In interface design asymmetry can keep you guessing and surprised rather than informed and aware. The previous two columns[3, 4] have explored the notion that the essence of an effective interface lies in being concise, consistent and conventional. Symmetry is a form of consistency.

JUnit[5], the popular open-source unit testing framework for Java, provides a number of assertion methods to allow programmers to test expected against actual values. There is an `assertTrue` method; there is also an `assertFalse` method. There is an `assertNull` method; there is also an `assertNotNull` method. There is an `assertSame` method; there is also an `assertNotSame` method. There is an `assertEquals` method; there is no `assertNotEquals` method. Oh. A hole in the whole. Jarring and inconvenient.

Such an inconsistency is annoying and tends to catch programmers out at one time or another. Of course, it is possible to work around it by performing the relevant equality comparison and passing the result to `assertFalse`. However, not only does this make a set of tests look inconsistent, it can set up the expectation that the normal way to work with an interface is to workaround its limitations – I certainly know APIs for which this is true, but their existence is grounds for caution, not a lead for recommended practice. It is true that one simple assertion is the absolute minimum that you need[6], but the whole point of a framework is that it captures and expresses the common features of your domain.

**Facts at a glance**

- A symmetric pair of methods represent some opposite aspect.
- The completeness of an interface can sometimes be considered in terms of its symmetry.
- Symmetry, or its absence, can affect ease of use.
- But not everything is symmetric – that would be forced and boring.

For a programmer to provide `assertNotEquals` in JUnit would not be complicated, but it would be tedious, and this is perhaps one of the reasons it does not yet exist. In truth, `assertEquals` is not a method: it is a family of overloaded methods, and quite a large family at that. Because of limitations in the existing Java language – contrary to market hype, Java is not a fully or consistently object-oriented language – a variant of `assertEquals` is provided for each built-in integral type (where == is used for the comparison), for each floating-point type (where comparison is with respect to a delta rather than strict equality) and for object types (where the `equals` method is used). So, although an intuitive interface to use, it is a shopping list to implement and document, involving no small amount of copy-and-paste duplication.

To provide `assertNotEquals` would involve providing the same number of overloads again, along with the associated implementation, documentation and duplication. One could make a case that this would bloat the interface and make life for the implementer more tedious. It is certainly not an exciting task, but the question is on which side of the interface boundary should the slog and repetition occur? Is it the implementer's responsibility or the interface user's problem? Put like that, it seems clear that the convenience of an interface should favour its user over its implementer – write once, right for many.

What about the question of bloat? One of the guidelines for effective interface design is that an interface should be concise. However, concise is not necessarily the same concept as short: to be concise means to be both brief and comprehensive. The measure of an interface is not necessarily in the number of methods it defines, but in the number of ways it can be used. So, although there are many `assertEquals` overloads, there is only really one `assertEquals` concept and only a couple of ways of calling it, so the usage is simple even if the actual Java method listing is long.

Taken from this perspective it is clear that the omission of `assertNotEquals` is more of a pain than its potential inclusion. Taken over time, it takes longer to discover and explain its absence than to include and use it. Note that little or no explanation is required for such opposites: if you know what `assertTrue` does, then you know what `assertFalse` does; if you are told what `assertSame` does, you don't need to be told what `assertNotSame` does.

It is interesting to note that as JUnit has evolved, its assertion interface has become more symmetric. JUnit version 3.8 introduced `assertFalse` to pair with `assertTrue`, an obvious completion. `assertTrue` was introduced in version 3.7 to replace the `assert` method, which clashed with the keyword of the same name introduced in Java 1.4. However, even before Java 1.4 forced the issue in favour of a naming convention that threw the question of symmetry into sharp relief, an astute interface designer might have pondered the absence of `assertNot` as a partner for `assert`.

Of course, not every assertion has an implied symmetry: JUnit's unconditional assertion, `fail`, has no obvious complement. The absence of failure defines success – likewise, "if we do not succeed, then we run the risk of failure", to quote Dan Quayle, the former US vice president, insightful logician and orthographer extraordinaire. Providing, or indeed requiring, a `succeed` method would run the risk of being both odd and awkward.

Looking back at the Java `RecentlyUsedList` interface described in the previous two columns[3,4], the most obvious asymmetry is the absence of an `isFull` query:

```
interface RecentlyUsedList
{
    boolean isEmpty();
    int size();
    int capacity();
    void clear();
    void push(String newHead);
    String elementAt(int index);
    ...
}
```

Just as `list.isEmpty()` more accurately captures the essence of `list.size() == 0`, `list.isFull()` would better express the intent of `list.size() == list.capacity()`.

The ordering discipline encapsulated by a recently used list ensures that the most recently pushed is at the head and the least recently pushed value is last. The use of the `push` terminology suggests perhaps a `pop` method could be useful, losing the least recent entry.

Whether or not one chooses to include these additional features, for convenience or capability, symmetry has certainly helped highlight them as considerations:

```
interface RecentlyUsedList
{
    boolean isEmpty();
    boolean isFull();
    int size();
    int capacity();
    void clear();
    void push(String newHead);
    String pop();
    String elementAt(int index);
    ...
}
```

The symmetry between `push` and `pop` is emphasised in their respective input and output. However, symmetry in code tends to be local, informal, evocative and sketched; it is rarely a global, formal, binding, mirror-perfect relationship. For example, there is no precondition on `push`, but `pop` cannot be called on an empty list; a successful `pop` always changes the visible state of the list, whereas a `push` of the most recently used item will leave the list unchanged.

## Reasonable asymmetry

Of course, it is easy to become overly enchanted by symmetry. For example, having a habit or working to a guideline to provide a *setter* with every *getter*. Not only is this not a good idea, it is actually quite a bad one[7]. Restricting object mutability either to full immutability – whether by copy, by class or by `const`[8] – or to a few well-defined and well-motivated state-modifying methods is a more reasoned and less error-prone path to take.

It is possible to query an entry in the recently used list by index, but not to set one by index. This is quite reasonable: setting a value directly would defeat the ordering discipline that is the whole point of a recently used list. It also leads to some interesting questions concerning one of the other invariants that govern the data structure: entries are unique. What would happen to the data structure if an entry were set, by index, to the value of an existing entry? There are many possible interpretations and responses, but few are worth pursuing. Such behaviour is outside the scope of a recently used list. The perceived requirement demands either a revisit or another solution. Scope creep often arises because such questions are answered in code rather than unasked.

## A more reasonable symmetry

Conversely, a reasonable guideline is that if you have a setter, there should probably be a corresponding getter. To be more precise, if an object has an attribute, either conceptual or physical, that can be modified by a method, there should be some way of querying that attribute. Failure to do so leads to write-only attributes, which can be frustrating to use and awkward – or impossible – to test.

For (counter)example, Java's `ArrayList` class has an implied attribute that describes its current capacity, which is how many elements it can hold without automatically resizing, as opposed to its current size, which is the number of elements it currently holds. The capacity defaults to 10 on creation, unless it is specified directly as a constructor argument, and can be specified explicitly using the `ensureCapacity` method or trimmed to the current required size using `trimToSize`.

The documentation talks about the capacity and how to change it, and the concept is visible in the interface, but there is no way to find out what it is. This means that the capacity is modifiable but unobservable, so the capacity-related operations are untestable: how would one write an assertion to confirm that the default constructed capacity was in fact 10? This may at first seem academic: testing the correctness of `ArrayList` is surely Sun's problem, so there is no need to worry about the issue. However, it is the design principle at stake, not the specific example: there ought to be a test somewhere – within or without Sun – so what would it be?

It is interesting to note that `ArrayList`'s less sophisticated, less well reasoned and less focused predecessor, `Vector`, offers the capacity query in question.

Write-only attributes can be quite subtle, and it is easy to miss them, especially if you are the implementer. Proximity to the implementation can make it hard to share the interface user's perspective. A critical eye – yours or someone else's – or a test-driven approach[6,9] can help to tune into this line of symmetry. ■

## References

1. Kevlin Henney, "Five Possible Things Before Breakfast", artima.com, June 2003, www.artima.com/weblogs/viewpost.jsp?thread=5432
2. Kevlin Henney, "Factory and Disposal Methods", VikingPLoP 2003, September 2003, available from www.curbralan.com
3. Kevlin Henney, "Form Follows Function", *App Dev Advisor*, March 2004.
4. Kevlin Henney, "Conventional and Reasonable", *App Dev Advisor*, May 2004.
5. JUnit, available from http://junit.org
6. Kevlin Henney, "Put to the Test", *App Dev Advisor*, November 2002.
7. Kevlin Henney, "The Good, the Bad, and the Koyaanisqatsi", VikingPLoP 2003, September 2003, available from www.curbralan.com
8. Kevlin Henney, "Objects of Value", *App Dev Advisor*, November 2003.
9. Kent Beck, *Test-Driven Development by Example*, Addison-Wesley, 2003.