Defining an interface is more than just putting a front on an implementation. **Kevlin Henney** looks into the intentional side of interface design

Form follows function

IT IS OFTEN SAID THAT GOOD DESIGN IS INVISIBLE. What does this mean for something like an interface, which is intentionally a highly visible boundary separating different parts of a system? Interfaces are about communication, both between different components of the system and between the code and the user. A well-designed interface is typically concise, consistent and conventional. Which all sounds a bit dull at first sight, but that is, after all, what it means to be invisible: there is little that is out of the ordinary and little to write home about.

Given the significant role that interfaces play in a system – whether we are talking about the interface presented by a header file, a class's public section, the `interface` construct found in Java, C#, IDL and other languages, a package's public section or even a whole API – the "principle of least astonishment" is perhaps one of the most important guidelines to follow:

- Concise: Interfaces that try to be helpful by accommodating all possible needs in all possible ways are particularly unhelpful and unaccommodating. An interface should be minimal, but not to the point of being cryptic. The less there is to read, learn and remember, the easier an interface will be to use.
- Consistent: An interface that is consistent in its partitioning and naming is easier to anticipate and offers fewer surprises than one that seems to be designed in a fragmented and disconnected fashion, following one style in one part of the interface and another style in a different part of the interface.
- Conventional: In selecting names, partitioning method and class responsibility, and organising argument lists and return values, follow established practice where such practice is itself concise and consistent – don't invent new styles or follow obscure ones unnecessarily, but don't slavishly follow common but poor forms. An interface that follows the idioms of the language and environment will be more readily grasped than one that does not.

Nevertheless, being concise, consistent and conventional is not quite as dull as it first appears: it simply shifts the emphasis and effort of interface design to more weighty matters, and away from the trivial. The use of idioms (expressions and phrasing specific to a cultural grouping) is in contrast to the use of idiolects (affectations of expression peculiar to an individual), wresting responsibility for certain possible, arbitrary variations away from interface designers, and freeing them up to focus on other design issues. The responsibility placed on the designer becomes more one of judgement and breadth of knowledge than one of invention.

Signature contracts

Interfaces play the role of contracts between different parts of a system[1], and there are many sides to drawing up such a contract[2]. The appearance and basic use of an interface constitutes a signature contract: names, types and argument lists. Developers familiar with the contract metaphor often overlook contracts that are not functional contracts, and they often assume that functional contracts are always – and can always – be expressed using only pre- and postconditions[3].

Most generally, a contract is an agreement that is binding and typically enforceable, with consequences for any transgression. The use of assertions to enforce correctness of usage and implementation allows some contractual aspects to be expressed, but it does not cover everything. In statically typed languages, signature contracts tend to be enforced at compile time: incorrect usage leads to a compilation error. In dynamically typed languages, or via reflection, signature contracts are checked at runtime: transgressions are signalled by exceptions. When going through weakly typed interfaces, such as via `void *` in C, any requirements on a type that are not met will lead to undefined behaviour – a laissez faire enforcement policy with crashing consequences.

Style and idiom

Consider a simple sorting facility, along the lines discussed in a previous column[2]. Narrowing the scope, consider sorting an array or subarray of integers. What would be the most idiomatic way of presenting this facility in C?

```
void sort(int array[], size_t how_many);
```

And in C++?

```
void sort(int *begin, int *end);
```

And in Java?

```
...
public static void sort(int[] array, int from, int to) ...
...
```

In each case the operation is named sort and returns void. However, in spite of the syntactic similarities between the languages, all other details are different. Although it is sometimes possible to adopt one language's idiom in another, this often leads to an inappropriate look and feel in the borrowing language. The style may work in a technical sense – it compiles and runs – but it may not make the best use of language features or communicate most effectively – it is more idiolectic than idiomatic.

For example, although it is possible to adopt the C-style signature for sort in C++, the native C++ approach for expressing algorithms is based on iterator ranges, as used through the Standard Template Library (STL). If the type of the underlying container is generalised to allow any random-access sequence, not just an array, or the type of the elements to be sorted can be anything that can be ordered using the `<` operator, the `sort` function generalises directly to the common function template appearance of the STL algorithmic operations:

```
template<typename iterator>
void sort(iterator begin, iterator end);
```

Each of the suggested signatures reflects a similar set of capabilities: the ability to take an array of integers and sort all or part of it. In C the subrange is communicated by passing a pointer to the starting element and the count of how many elements after that should be considered, expressed as the standard `size_t` type, used for denoting sizes, rather than a plain `int`. This signature takes advantage of C's close (sometimes too close) relationship between pointers and arrays. By contrast, C++'s notion of subrange is based on an iterator that points to the beginning and one to the first element past the end of the subrange.

Java does not allow pointer–array aliasing, so although its signature is similar to the C version, it must name the base of the array explicitly. Java then specifies the subrange using integer indices. Like the C++ version, its range is half-open, i.e. the first index is included in the range but the last is not. An alternative would have been to specify the start followed by the number of elements to be sorted rather than the start and the end indices. However, this is not the convention in common use elsewhere in the Java libraries, so although it has its own merits, those are not enough to outweigh the use of the more idiomatic form.

In C and C++ the operation name is global and, in C++, perhaps nested within a namespace. In Java there is no mechanism for defining global operations, so sort needs to be defined in the context of a utility class, which raises new questions. What should be the granularity and scope of such a utility class? A class, `Sort`, that holds only the single `static sort` method; a class, `Sorting`, that holds sort and any related methods, such as an `isSorted` predicate; a class, named something like `ArrayUtilities`, that holds `sort` and other unrelated methods that operate on arrays. The first option is too fine grained and offers no useful room for expansion, with the exception of adding other sort overloads. The third option is the one used by the Java Collections API, and should be avoided as being little more than a coincidental and un-cohesive aggregation of functionality. The second option is probably the wisest in this case.

Useful, usable and recently used

There were few naming issues in considering the sorting facility: sort was sufficiently concise and conventional. The stylistic concerns were focused mainly on the argument list. Let's take another example that demonstrates a broader interface to a commonly used facility, but one that is less frequently commoditised than sorting: a recently used list.

Application menus often hold a list of the most recently opened documents and modern phones normally hold a list of the most recently dialled numbers. A recently used list is often bounded to an upper capacity. The general behaviour is that of a dispenser type, such as a stack or a priority queue, with the property that the most recently inserted element is found at the front and that an element occurs only once in the whole sequence, i.e. no element is equal to any other element.

To keep things simple, rather than focus on holding arbitrary objects that may be mutable and could undermine the uniqueness invariant on the list, just consider the interface to a recently used list that holds strings. What would this look like in Java?

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

This interface follows standard Java capitalisation conventions and uses names found in the Java library. In the Java Collections API the name `isEmpty` is favoured over the less commonly used `empty` convention, which would be the appropriate name for such an operation in C++. Notice that the common but rather lame `get` prefix convention is avoided in naming `size` and `capacity`. This prefixing convention is over-applied by many programmers, with the typical effect of making their code look like a form of high-level assembler. Joshua Bloch intentionally avoided this style in the design of much of the Java Collections API [4], favouring the clearer and more direct adjective style for queries about properties.

There is some inconsistency in Java over what to call the size or length query: arrays use a `length` field; strings use a `length` method; collections use a `size` method. As a `RecentlyUsedList` represents a collection, it makes sense to adopt `size`. The most related notion to an upper limit is the `capacity` method found in `Vector`, which also has the virtue of being a concise name. There is little contention or uncertainty in naming the `clear` method, which is the conventional form used in Java and other languages.

If a recently used list is in some ways like a queue or a stack, common received wisdom suggests that `push` is the most likely candidate name for the insertion operation. Java's `Stack` class does indeed use push, but this legacy class is a little bit of a design mess – for example, its inappropriate inheritance from `Vector` – so it is not necessarily the best example to cite. The signature of

Stack's push is also slightly different, returning the newly inserted element rather than void. Not a terribly useful or common feature, so one that has been omitted in RecentlyUsedList.

A recently used list is not much use if it is a write-only collection: you need to get your hands on the contents if you want to be able to display them. Although drawn from the legacy Vector interface, the elementAt name has the virtue of being clear in its intent. The alternative name, found in the Java Collections API, is the terser and somewhat less intentional get.

Another alternative, again from the Java Collections API, would be to rename push as add. This name is less specific because it is supposed to apply to any Collection implementation, so add is a more neutral term than push for an interface that can be implemented as a sequence or a set. The following is an alternative interface that strikes a slightly different balance between conciseness, consistency and the conventional:

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

If RecentlyUsedList were to be integrated into the Collection hierarchy, and generalised to handle Object not just String, this would be the preferred form. However, the form presented first is more specific and a little clearer in its intent, given the current scope of the type's design.

In either case, it is just worth noting that the integer index to elementAt or get would count from zero and not one. This may seem obvious, because Java is a zero-based language for array and collection indexing, but some programmers may be tempted to start from one because the display presentation of recently used lists is almost always one based. The representation within the program should follow the conventions of the language – zero counting – and should not be coupled to the presentation logic – one counting.

OK, so these are the considerations that go into expressing the interface in Java. How about C# or C++? And what about consistency and completeness within an interface? More on these questions next time. ■

References

1. Butler W Lampson, "Hints for computer system design", *Operating Systems Review*, October 1983.
2. Kevlin Henney, "Sorted", *Application Development Advisor*, July 2003.
3. Bertrand Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.
4. Joshua Bloch, *Effective Java*, Addison-Wesley, 2001.