Requirements are hard enough without throwing around vague phrases like *non-functional*. **Kevlin Henney** sorts out the language of requirements

Inside requirements

THINKING IS NOT SOMETHING THAT CAN BE reasonably redistributed or outsourced, but this is not to say that people haven't tried. Programmers are knowledge workers, which means that it's important to know stuff. Following an observation from Alan O'Callaghan, former *ADA* columnist, programming is applied thought, which means that programmers need to think about stuff.

So, what is it that programmers need to think about? Many, many things, but let's concentrate on requirements. Requirements come in many possible flavours, but are commonly cast into two categories: functional and non-functional requirements. As a label, it has to be admitted that *non-functional* is fairly lame. It is unhelpfully vague and amusingly ambiguous.

Most things that are non-functional don't work: washing machines, cars and programs that are non-functional are broken. Also, by prefixing *functional requirements* with *non*, other requirements seem to be relegated to second- or third-class citizenship.

Requirements can be better and more fairly considered under the headings of functional requirements, operational requirements and developmental requirements. The requirements that programmers must consider are not just the customer-centric ones specified for the application as a whole. As a problem-solving activity programming is recursive, which means that requirements at the level of the application generate requirements at the level of the component or subsystem, and so on down.

Functional requirements

Functional requirements focus on purpose. They collectively define what a system does, typically in terms of visible changes that you can effect in the system or that it can cause in the outside world. These shared phenomena are governed by rules, and are typically deterministic — how interest accrues over time, how a shopping basket is totalled up, how time-series measurement data is ordered.

Functional requirements are typically binary in nature: you either meet a requirement or you don't. There are no shades of grey, no extra-Boolean possibilities lurking between *true* and *false*. It either passed or it failed.

That's the nice thing about functional requirements. They are precise. There is no approximation. They are easy in the sense that it's easy to tell whether or not they have been implemented correctly. If a functional requirement is not met precisely, you have a bug. Some bugs are systemic, but many functional-requirement bugs can be traced to a single expression or a handful of statements.

All this means that functional requirements can be tested directly, objectively and, as a result, automatically. This applies from the level of a whole system all the way down to its individual methods. Hence the idea that unit tests can repeatedly, automatically and meaningfully pass judgement on a component [1, 2, 3].

The common expression of functional requirements at the class level is in terms of a contract [4]. An interface establishes a contract between its implementer and its user. The most common perception of this contract is in terms of preconditions and postconditions on methods, describing what is required of the caller before a method is called and what is guaranteed as a result [1, 5]. However, this is not always effective or possible. Callbacks, or any inversion or discontinuity of control flow such as multithreading, do not sit well with just the pre- and postcondition model [6]. Some functional requirements can be expressed more simply by other means. For example, the `equals` method in

FACTS AT A GLANCE

- **Requirements inform and shape a system, which means that although functional requirements are necessary, they are not sufficient.**
- **Functional requirements focus on a system's purpose and are, in principle, automatically testable.**
- **Operational requirements focus on a system's quality of service.**
- **Developmental requirements focus on a system's quality of implementation and are the preserve of the programmer.**

Java, used for implementing an equivalence relation between one object and another, is defined in more mathematical terms. The requirements of reflexiveness, symmetry, transitivity, consistency and null inequality differentiate correct implementations from incorrect ones. Unit test cases can also be considered to explore a contract by example and in executable form.

Ultimately all code is a fiction, but a fiction shaped by expectations. Programs exist in an artificial universe, fabricated for function, created for a purpose. The functional requirements relate the story and move the plot forward. But a good story needs more than just a decent plot: it needs style. A lot of it is in the telling. This is where requirements other than functional come into play.

Operational requirements

Operational requirements focus on how a system achieves its functional requirements. Operational requirements — and the ability too meet them — determine the quality of the user experience. Functionality forms part of the user experience, but inasmuch as functional requirements are satisfied, they do not uniquely determine the quality of the experience. Of course, if the application is buggy, the user experience will be somewhat less than stellar.

Under the umbrella of operational requirements, you can find performance, throughput, memory footprint, scalability, usability, availability, manageability, etc. These quality-of-service aspects are continuous — even fuzzy — rather than binary in nature, often lacking the concrete objectivity of functional requirements. Application usability is somewhat harder to quantify than the desired effect of a cash withdrawal on your account. Likewise, 99.49% availability in a given week against a requirement of 99.5% is not necessarily an outright failure. It amounts to around a minute in 24x7 period, which is likely to be less significant than the accumulated standard error in the measurement.

The subjective nature of usability means that automated testing is an impossibility. The statistical nature of many other qualities means that deeper analysis of results is required before conclusions can reasonably be drawn.

Unlike functional requirements, operational requirements cut across the code structure. Where functional requirements can be fulfilled in a modular fashion, e.g. by decomposition into classes, operational qualities are typically emergent in nature. This is what makes them harder to assess and to design for.

Operational questions draw a programmer into a more complex engagement. Prototyping can be used to demonstrate both user interfaces and the performance (or scaling, or memory usage, or...) viability of a particular proposed design.

The absence of a will to prototype is a common problem and can lead to architecture by guesswork. For instance, the introduction of threading into an application is often a reflexive response to operational requirements that call for responsiveness or throughput. However, sometimes threading is a problem to a solution rather than the other way around. Multithreading is not a drop-in facility. If applied incautiously it can slow an application down rather than speed it up. Worse, threading can introduce shy bugs, which come out only once in a blue moon.

Operational requirements can also form part of a class's contract. Of course, such contractual undertakings do not fit the tidy Boolean universe of pre- and postconditions. For example, C++'s STL is strong on performance complexity contracts for functions. `std::sort` of *N* elements guarantees an average complexity of *N log N*, giving the functional guarantee that all the elements will be in ascending order. `std::stable_sort` guarantees an upper limit of complexity of *N* (log *N*)^2 or, if enough memory is available, *N* log *N*, with the stronger functional guarantee that not only are the elements in ascending order, but the ordering of equivalent elements is also preserved.

The subscript operator, `operator[]`, on STL sequences is guaranteed to be present only if subscripting into the container takes constant time to perform the lookup. No such contract exists for the moral equivalent in Java. Calling `get` to access an element by its index may take constant time, in the case of an `ArrayList`, or it may be linear, in the case of a `LinkedList`. This means that a loop using `get` to visit each element of a 5000 item list will result in 5000 accesses on an `ArrayList`... and up to 12500000 accesses on a `LinkedList` as the trail of links is traversed and retraversed. It is good to separate concerns and encapsulate unimportant implementation details, but encapsulation does not call for smothering all operational qualities under the guise of abstraction. In a relatively recent and retrospective move a `RandomAccess` marker interface was added to the Java Collections API to allow (but not require) collection users and authors to differentiate between constant- and linear-time implementations of `get`.

Developmental requirements

If operational requirements define the user experience, developmental requirements are about the programmer experience. The *-ilities* of interest are portability, comprehensibility, changeability, etc. These quality-of-implementation aspects do not necessarily affect functional and operational qualities directly, but over time they can drag on them.

As we have just been talking about performance complexity, it seems appropriate to point out that changes to well-factored and clear code tend to follow a linear cost curve over time, but code that is left to rot and accumulate changes tends to follow a square law. The resistance of such code to successful change increases disproportionately over time, so that it can eventually reach a steady state: as many problems are introduced as are fixed.

Where functional and operational requirements are the preserve of the customer, developmental requirements fall to the programmer. This means that they are often neglected, more so than operational requirements, because they are not directly visible to or demanded by the customer. However, it is the job of the programmer to address these. This is part the knowledge that any domain expert is expected to bring to the table; it is not something to be prompted for. ■

References

1. Andrew Hunt and David Thomas, *The Pragmatic Programmer*, Addison-Wesley, 2000
2. Kent Beck, *Test-Driven Development*, Addison-Wesley, 2003
3. Kevlin Henney, "Put to the test", *Application Development Advisor*, November–December 2002
4. Butler W Lampson, "Hints for computer system design", *Operating Systems Review*, October 1983
5. Bertrand Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997
6. Clemens Szyperski, *Component Software*, Addison-Wesley, 1998

Kevlin Henney is an independent software development consultant and trainer. He can be reached at http://www.curbralan.com