Requirements and Specifications

Allan Kelly, allan@allankelly.net

For many, perhaps most, development teams the terms *requirement* and *specification* are used interchangeably with no detrimental effect. However it is sometimes useful, and occasionally important, to differentiate between the two terms:

> "A requirement is a desired relationship among phenomena of the environment of a system, to be brought about by the hardware/software machine that will be constructed and installed in the environment. A specification describes machine behaviour sufficient to achieve the requirement. A specification is a restricted kind of requirement" (Jackson and Zave 1995)

The key points to note are in the last two sentences:

- A specification describes behaviour to achieve requirement
- Specification is a restricted requirement, i.e. the specification narrows down the requirement

For example: there may be a *requirement* to store customers details for shipping and future marketing. The *specification* would state what details should be stored (e.g. name, postal address, e-mail address, etc.). Specifications can be very detailed, e.g. a postal address should contain an house number or name, a street, a post code, and the format the post code should satisfy.

Specifications

In creating the specification the requirement may change. For example: should the system accept US style zip codes as well as UK style postal code? This depends on whether the system is required to service UK only customers, this might mean those commissioning the system need to consider their international approach.

In exposing the detail of the specification the requirement may be brought into question, refined and even changed.

There is almost no limit to the detail a specification can reach. In University I was taught to write incredibly detailed specifications in formal, mathematical, logical notation. Yet for many teams this level of detail is unnecessary.

For many teams the specifications are uncovered as part of the coding process. Indeed code itself represents the ultimate specification of what happens, unfortunately in this form the specification is difficult for non-programmers to understand and therefore agree and verify.

In some fields leaving the specification to the programmers is a good thing. Programmers who understand the field may have little need of additional (expensive) documentation; in fast changing environments writing down a specification and communicating may injected undesirable delays.

In other fields it is preferable to have specification understood in advance or determined by specialists. For example competitor organizations may agree on a specification for completing products in order to facilitate interoperability.

For teams working in traditional - upfront requirements, specification and design - specifications can become a battle ground. Programmers put under pressure, without knowledge or specifications, inevitably do things the consuming clients do not expect. One side will demand more detail next time to prevent the problem.

But more detail doesn't solve the problem because a) nobody remembers the details, b) omissions and mistakes are made in specifying the detail, c) more detail leads to more things that can change, more things to be read (and forgotten) and more opportunities for mistakes in the detail

Since the amount of detail is almost infinite the call "for more detail" easily escalates into an arms-race. Introducing more detail in specifications can easily make things worse not better.

When working in short iterations requirements are best given at the start of each iteration - not all requirements need to be known in advance but enough for the duration of the iteration should be. (Teams which embrace unplanned work can happily start an iteration with missing requirements or respond to unexpected requirements. Teams which aim for maximum predictability will see unplanned work as problematic and aim to pin requirements down in advance.)

Specifications on the other hand might be known in advance or might be discovered during the iteration, either as a specific exercise or as part of the coding activity. Sometime leaving programmers to finalise specification is not only possible but desirable. Other times specifications might be determined in advance by a specialist, typically an analyst of some description.

Problems occur when specification are decided far in advance. When this is done specifications decay because:

- A changing world leads to specifications, and requirements, to change too.
- As the development team create the system they learn about both problem and solution domain, this can lead them to new insights.
- Without a deadline analysis can continue almost indefinitely, allowing more time for work to occur most likely leads to more work.
- The more specifications are decided the more that can change.

One source puts the rate of requirements change at 2% per month during design and coding (Jones 2008), which works out at an annual compound rate close to 27%. Changed requirements means changes specifications which leads to rework.

Therefore it is preferable to decide specifications as late in the day as possible - say in the same sprint as coding will occur, or in the previous sprint. In the early descriptions of Behaviour Driven Development (BDD) Dan North described a Business Analyst working at the same keyboard as a programmer writing specifications as code was written.

And tests

A development story, especially when in User Story format (Cohn 2004), is usually a requirement. It is a token for work to be done and is often called *a placeholder for a conversation*. The acceptance criteria often found on the back of a story card are specification but rarely are they a complete specification. Detail can be pinned down later in a conversation.

In some teams a fuller specification will be created in the form of acceptance criteria produced by a requirements engineer or professional tester before coding begins. If this is not done then the specification will be completed at the time of coding or at the time of testing.

Difference in interpretation of requirements and specification by programmers and testers is a common source of bugs. Two individuals read the same document(s), the programmer interprets it one way and writes code as such; the professional tester interprets it differently and tests it as such.

The most detailed specifications actually move beyond acceptance criteria to become acceptance test scripts. When these are written in natural language there is room for ambiguity. When written in a formal form ambiguity is squeezed from the system. When the formal form is executable - such as a FIT table or Gerkin *given when then* then it is possible to ensure the program code and specification match. This is an executable specification.

Although ambiguity may be squeezed out of specifications by formalising them it is more difficult to eliminate omissions. To extend the above example, tests may be used to ensure an address postcode matches the prescribed format but tests cannot ensure customers supply their county unless a human intervenes to specify county as a necessary field.

How much detail specifications and tests need to specify, and the point at which the details are decided varies greatly. For some teams specification can be left in the hands of the programmer when they are coding. In other environments specifications needed to be pinned down by specialists well in advance.

However whenever specifications and tests are to be used they should be created before coding begins. Too create them after the programmer has completed their work is to invite discrepancies and rework.

Automated acceptance tests, the new formal methods

Automated acceptance tests, also known as executable specifications, continue the tradition of formal logic specifications. The tests are a specification and automation demands formalisation. The first difference is that Gerkin style *given when then* specifications, for example, are readable by most people while VDM-SL predicate logic is only meaningful to those with years of experience.

It is interesting to note that the *give when then* specification format mirrors the pre-post conditions used in formal languages like VDM-SL. The *given* declares a set of pre-conditions and the *then* declares the post-conditions.

Secondly both techniques require tool support to be effective. But while predicate logic specifications tools are few and far between, expensive and difficult to use the tools used for executable specification are largely available free of charge as open source, e.g. Selenium, JBehave, Cucumber and FIT/FITNESSE.

Rather than providing a logical description of the program under development (as the likes of VDM-SL do) these tools work through examples. Specifications are given by way of examples - hence the name *specification by example*.

Because these examples are executable as tests it is possible to validate the program satisfies these specifications.

These examples may not be exhaustive, there may be undefined behaviour in the system where specification examples are not provided. The program code is still the ultimate specification, the examples aim to cover observable behaviour where it is of significance to those doing the specification. (This is differs from predicate logic descriptions which may aim to be sound and complete).

Knowledge and Trust

Whether specifications are needed or not often boils down to knowledge and trust. When developers have extensive knowledge of the domain they are working in then much of the information that would be contained in specifications already exists inside their heads. And when they have ready access to others who know more about the domain a verbal conversation may substitute for a written document.

For example, while most programmers will be familiar with the postal address example above only those who work in specialist domains will know other formats. English legal practitioners frequently use DX mail rather than Royal Mail. Programmers who have worked in legal software for some years may automatically provide DX number and exchange in software while those new to the field need to be told, i.e. they need to given a specification, this may be written or verbal.

However knowledge alone may not be enough if programmers and testers are not trusted to use their own knowledge, or not able to ask for assistance when they recognise they do not.

Forgoing specification documents saves money because documentation is expensive. It also accelerates development because writing documentation is a time consuming process prone to blocking. Having a programmer determine specification as they code is the ultimate in just-in-time working.

Still there may be merit in having another person bring their knowledge and understanding to the specification effort. If these specifications are to match the code they must be created in some fashion which minimise opportunities for differences in understanding to emerge.

On anything other than trivial systems using human effort to validate that specifications and program match becomes a time consuming and an error prone exercise, and thus slow and financially expensive. To overcome this

specifications need to be both machine readable and executable - through automated tests or theorem validation - to ensure code and test specification say the same thing.

When knowledge and trust are lacking specifications become necessary, so too does an effective, usually automated, means of validating code against specification.

Conclusion

- There is often little point in differentiating between *Requirements* and *Specification* and the two terms are often used to mean the same thing, i.e. the thing to be built.
- Sometimes it can be useful to consider specifications as distinct from requirements. In such cases specifications are a
- Requirements are best given at the start of each iteration but specifications can be discovered within the iteration. Finalising specifications as late as possible has a number of advantages.
- Requirements are unavoidably imprecise. Specifications should not be.
- Discovering specifications can lead to changes in the requirements. Requirements come before specifications but specifications can send ripples back to requirements.
- Specifications are test criteria; both specifications and test criteria can be formalised. Formalising specifications as predicate logic is time consuming and rarely justified. Formalising tests as executable specifications can be highly effective.

About the author

Allan Kelly has held just about every job in the software world, from system admin to development manager by way of programmer and product manager. Today he works helping teams adopt and deepen Agile practices, and writing far too much.

Allan works for Software Strategy (http://softwarestrategy.co.uk/) providing consulting and training to help teams adopt and deepen agile practices. He specialises in working with software product companies and aligning products and processes with company strategy. While agile is most closely associated with programming side of development he finds that the "what are we building" question needs to be addressed just as much - if not more!

He is the author of two complete books: "Business Patterns for Software Developers" and "Changing Software Development: Learning to be Agile", and one work in progress "Xanpan - reflections on agile and software development" (https://leanpub.com/xanpan). He is also the the originator of Retrospective Dialogue Sheets (http://www.dialoguesheets.com).

More about Allan at http://www.allankelly.net and on Twitter as @allankellynet (http://twitter.com/allankellynet).

References

Cohn, M. 2004. *User Stories Applied*. Addison-Wesley.

Jackson, Michael, and Pamela Zave. 1995. "Deriving specifications from requirements: an example." In *Proceedings of the 17th international conference on Software engineering*. Seattle, Washington, United States: ACM. doi:http://doi.acm.org/10.1145/225014.225016. http://mcs.open.ac.uk/mj665/Icse17rc.pdf.

Jones, C. 2008. *Applied Software Measurement*. McGraw Hill.