Something old, something new: 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. In everyday development conversations the terms are used synonymously, one is as likely to for the "spec" as the "requirements."

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 only UK customers. Consequently those commissioning the system might need to consider their international approach.

In exposing the detail of the specification the requirement may be brought into question, refined and even changed. A question of detail may ripple all the way up to the strategic level. Although, as with good code, one hopes that such ripples will not occur that often. If questions arising from specifications regularly ripple back to it may be a sign that the requirements, encapsulations or even strategy and goals are weak.

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 called VDM-SL (C. B. Jones 1986). Yet for many teams this level of detail is unnecessary and for most teams is not economically justified.

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 to and, more importantly, 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 the specification understood in advance or determined by specialists. Competitor organizations may agree on specifications to allow completing products to interoperate. For example, passing short SMS text message between competing handsets over competing networks using competing switching equipment requires all parties to follow agreed specifications.

For teams working in a 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 or other 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 quickly make things worse not better.

For example

Consider the CEO of a large super market change. His strategy is for increased market share. He, and his board, is prepared to trade profits for market share. The requirement he gives his COO is: increase sales.

Given this requirement the COO convenes his team. They determine that some B2G1F offers - Buy two get one free - are called for. They task the marketing team with deciding which products to apply the offers to and the IT team with providing the systems to implement this.

The IT team receive the B2G1F requirement and quickly realise that one requirement is for the products it applies to must be configurable. But how configurable? Does the marketing team require a web interface? Or can it be managed through a XML config file? The original requirement expands into multiple small requirements.

Then there is a question of how the offer is implemented. Requirements become very specific. Obviously when a customer buys two identical products a third identical product is free. But what if the customer buys two of the products in large and a third small one? Is the third small? Or one of the large?

And when the marketing team say "1 litre fruit juices" does that mean that someone buying 1 litre each of orange juice, apple juice and cranberry juice gets one free? And does that mean the B2G1F offer needs to be marketed differently?

Requirements can be large, they can be small, they can hide details which later become significant. Over time requirements are refined, they are atomised and details added. At some point a requirement becomes a specification.

Enter the Iteration

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 advantageous. Other times specifications might be determined in advance by a specialist, typically an analyst of some description.

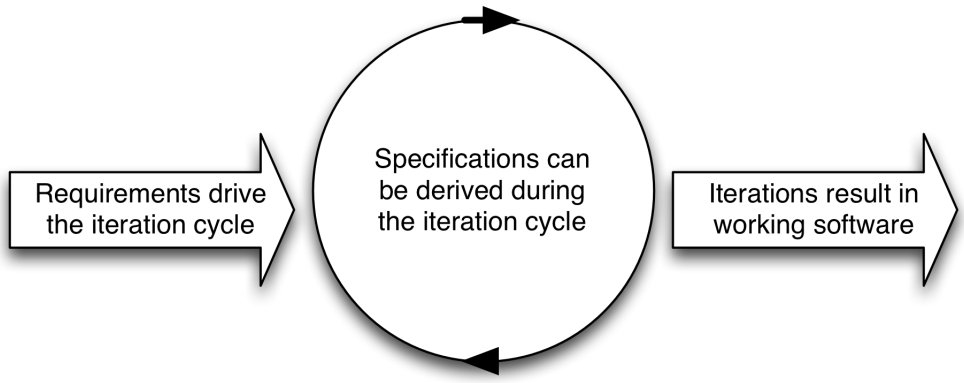

Figure 1: Requirements go in, working software comes out

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.
- Pre-emptively creating specifications will increase the amount of work done which is later discarded. Delaying specification creating reduces the chances that development work is cancelled, removed or changes after specifications are created.

One source puts the rate of requirements and specification change at 2% per month (C. 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.

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

Formal methods removed this problem by using exact logical notation but in doing so make the specifications difficult for novices to read and increase the chances of errors in the specification itself.

Another solution is to use acceptance test scripts derived from acceptance criteria as the most detail form of specification. 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.

Whether in a logic based notation such as VDM-SL or pseudo-English Gerkin the aim is the same: a specification that can be executed by a machine. The difference is when this execution occurs:

- Gerkin style executable specifications are used as after code is written as tests to ensure the human programmer produces the right thing.

- Logic based formal specifications aim to direct the code itself, either by executing the logic directly or by machine translation to a language such as C.

Although ambiguity may be squeezed out of specifications by formalising them it is more difficult to eliminate omissions. To extend the earlier 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.

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. While this is not guaranteed to eliminate problems it can significantly reduce them.

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. Automation demands formalisation because automation requires code. 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 VDM-SL, Z, CSP and other formal methods do) these tools work through examples. Specifications are given by way of examples - hence the name

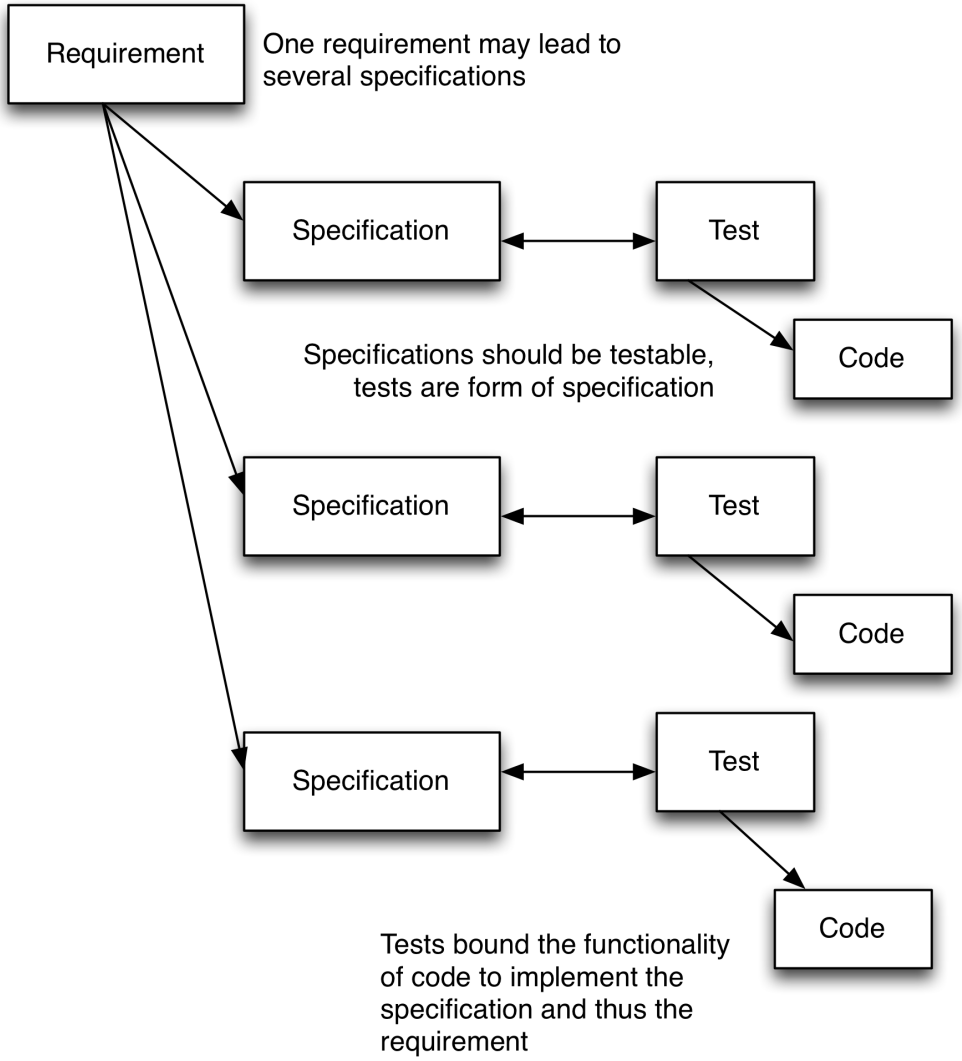

Figure 2: Specifications are derived from requirements and implemented as tests; and tests bound code

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 differs from predicate logic descriptions which aim to be complete).

Because BDD and ATDD tests are machine executable they are actually a form of formal specification. This author believes a useful line of research would be to see if such tests could be transformed in more traditional logical notations like VDM and Z

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 given previously, 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 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.

30 years ago the software industry attempted to solve the specification problem with formal methods. The BDD and ATDD techniques in use today take a similar approach but with a far lower barrier to entry. They in effect reinvent formal 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.
- Distinguishing between specifications and requirements can add to understanding.
- 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.

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.

Jones, C. B. 1986. *Systematic Software Development using VDM*.