# PRODUCT QUALITY THROUGH CHANGE MANAGEMENT

Richard Brooksby, Geodesic Systems, 1999–05–20

**Abstract**

Software development is all about increasing the value of our products to customers. This paper presents a method of planning, tracking, and managing change to a product, and of directing change at increasing that value. It describes evolutionary planning and delivery using a Perforce-based information system. The paper is based on the author's experience of introducing Capability Maturity Model level 2 and 3 key process areas in small (less than 20 engineer) software organizations.

Geodesic Systems <URL: http://www.geodesic.com/> is the leading supplier of enterprise memory and resource management solutions, tools, and services. Richard Brooksby <rb@geodesic.com> is the Vice President of Engineering and Development.

## 1. Introduction

In this paper[1] I'll tell you about the best methods I've found for managing change to a software product. The method I describe covers pretty much all of the key practices of level 2 of the Capability Maturity Model [CMM1.1].[2] It's a requirements-driven process, not specialized for any particular programming method.

## 2. The goal of software development

The goal of development is to increase the value of the product. The value is measured by the customers in the product's market. If your product is valuable enough to them they will pay you part of that value.

But our customers keep changing their minds about what's valuable. Software projects are faced with continually and rapidly changing requirements. A quality product must still meet those requirements, and it follows that a process that's going to produce a quality product must track them carefully.

## 3. Knowing what is valuable

To achieve the goal we must:

1. understand what is valuable,
2. make sure that our efforts are focused on what is valuable, then
3. deliver something more valuable than before.

Understanding what is valuable is the key to the whole process. It's achieved by "requirements management" [KPA1.1, L2–1].

[1] This paper is based on a Geodesic Systems internal document [RB98b].

[2] This document does not cover "software subcontract management" [KPA1.1, L2–43], and in any case this has been changed to "software aquisition management" in the draft CMM version 2.

Requirements management is how development identifies what the customer wants. The main function of requirements management is to maintain a document containing the customer requirements. This document states clearly what is desired by the customers, and should also place a value on each requirement estimating how much it is worth to the company. Here are some examples of requirements:

- a feature;
- an environment, for example having the product on Windows NT;
- compatibility with other software;
- a performance attribute, such as having the product process more than 10 widgets per second;
- an engineering attribute, such as needing less than one month to train a new developer to maintain the product;
- a reliability attribute, such as crashing less than once per week.

It's not necessary for the requirements document to be complete, provided it covers the things that are really important. The most important things to define are usually not the features, but the critical attributes. They're the things that are most difficult to track and control [Gilb88]. Start with what you know. You will quickly discover what else you need to say.

The requirements document contains our best idea of what the customer wants and values, stated in objective and measurable terms. "It works out of the box" isn't very measurable, but it can be broken down into measurable components: installation time, time to first use, ease of use, experience of first 30 minutes, etc. [Gilb88, chapter 9].

Whenever we discover something that the customer wants we add it to the requirements document. We then use the requirements to plan development.

## 4. Delivering value

The problem with the requirements is that they are always changing, and they usually recede into the distance, so they can never really be met. If we try to meet them all in one long delivery cycle we're bound to deliver something that is not wanted, or even deliver nothing at all until it's too late.

Attempting to meet all the requirements at once is called the *big bang delivery* [Gilb88, pp9–10]. The development group goes into hyperspace and hopes to come out somewhere near the desired result. The product is dismantled, and nothing works until the whole thing is done. A wasteful "code freeze" is needed to put things back together and "get it working". Furthermore, the product goes untried until near the end of the cycle, so that errors in requirements, design, or coding are not discovered until it's too late to do anything about them. It's an extremely risky strategy. Errors in requirements are most costly.

The solution to this problem is *evolutionary planning and delivery* [Gilb88, chapter. 7].

The idea is simple:

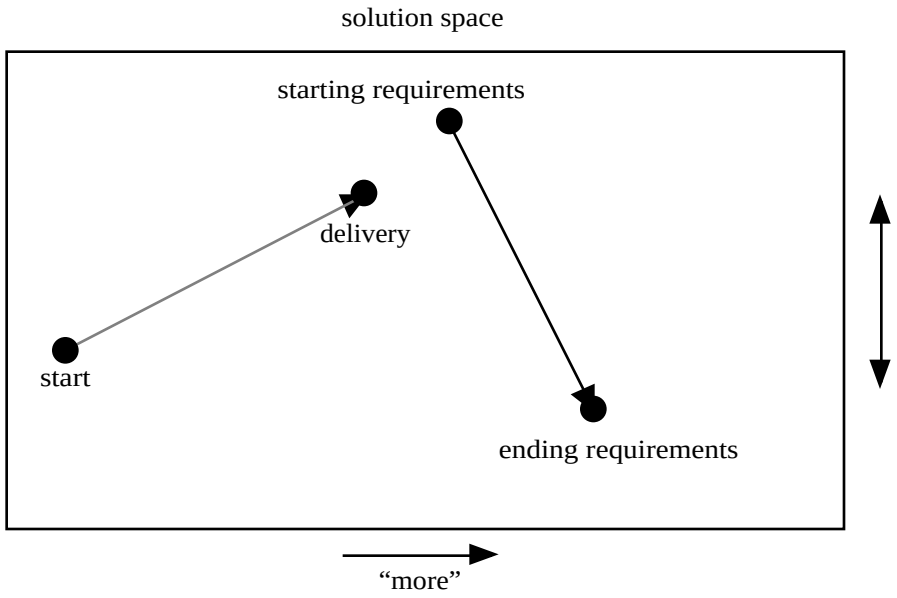



Figure 1. Big bang delivery

1. Compare the software now with the requirements, which are what we currently think the customer wants. This gives us a "sighting" — a direction to head in.
2. Take a step in that direction by tackling the biggest requirement or risk, delivering the maximum value to the customer in the shortest time.
3. Deliver a complete product (that is, fully packaged, documented, and so on). It doesn't do everything the customer wants, but it's much better than before.
4. Gather information from the customer, especially reactions to the delivered product. This will modify our idea of what the customer wanted (usually quite a lot).
5. Iterate.

This is called the *evolutionary delivery cycle*. Take a sighting, make a step, take another sighting, another step, and so on (see figure 2). We need to keep up with the changing requirements, of course, but we are much more likely to get close to what the customer really wants, even if the customer's ideas change. The shorter we can make the cycles the better for us.[3] There is less effort wasted going in the wrong direction and we deliver a more valuable product.

Actually, every programmer already knows about this process, because it's just like the "edit-compile-run" cycle. Programmers do some development, try out the result, modify their ideas, then do some more development, and so on. Here, we "edit" the product, "compile" it into a release, and "run" it by the customer.

The earlier we can make a release to the customer the better and more accurate our

[3] The ~100kloc projects I've managed did well on a two month cycle.

sightings will be, and therefore the higher the quality of the result. Customers are also

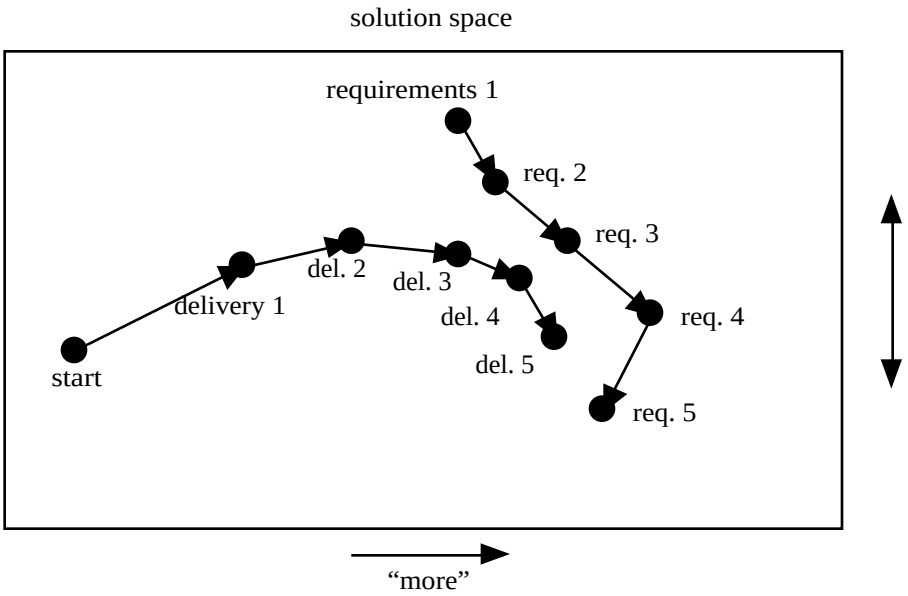



Figure 2. Evolutionary delivery

happy because they have software which meets some of their most important requirements sooner.

Another significant benefit of this technique is that it clears up important delivery issues early. Quite often there are critically important things that get forgotten until the last minute.[4] By delivering a complete product at each stage we discover these much earlier, and last minute panics are averted. Last minute work tends to be defective, and if it's something critical then that's even worse.

The disadvantage is that you have to do more analysis and planning up front, and it seems to take longer to make changes to the software that you "know" are needed. The advantage is that the final result is much closer to what's wanted, so you don't waste time on unnecessary changes or reworking it because you find out you didn't know after all. In my experience the advantages outweigh the disadvantages by a significant margin. The trouble is that you experience the disadvantages first.

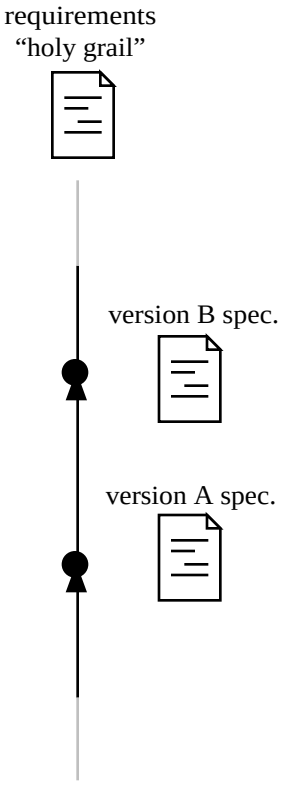
## 5. Planning to increase value

Once we have some requirements, the next step is to look at the current status of the software and compare it to the requirements. If you don't know the current status then it's both urgent and important that you develop ways of measuring how well you're meeting the critical requirements; you might be able to develop tests that do this, but some subjective measurements may also be needed.

There are bound to be differences between the status and the requirements. Resources are limited and you can't hope to meet all the requirements at once. Pick out the most

[4] Like the user manual.

valuable requirements that you don't meet at the moment. Give highest priority to the requirements that present the biggest risk of failure [Gilb88, chapter 6].




version plan

| | A | B | C |
|---|---|---|---|
| req. w | no | yes | yes |
| req. x | 50% | 90% | 95% |
| req. y | 30min | 20min | 20min |
| req. z | 3K | 5K | dynamic |

Figure 3. The development plan

Here's a practical example. A customer requires that the software runs on a new platform. This requirement needs some refinement: are all the features and all the performance required on the new platform? Which are most valuable to the customer? If you could only deliver one or two features on the platform which ones would they be? Put those first.

Get these high priority requirements analyzed so that you have some idea of the amount of effort required to meet them. Take every opportunity you can to divide them down into more manageable (smaller) pieces of work, and put the most effective pieces first.

You are now in a position to plan the next few versions of the product using *version planning*, a method of "software project planning" [KPA1.1, L2–11].

A *version* is a point in the evolution of the specification of the product. A version is a fixed subset of the requirements that you actually intend to meet, and meet with a schedule. You can think of a version as a snapshot of the ever-changing requirements.

The next version of the product should meet the high priority requirements that you selected earlier. The version after that should meet the next few, and so on. You should be able to work out when you can deliver the versions of the product because you have estimates of the work required. Keep the next version within easy reach, and make the

date of the next version as close as you can. If there's work that seems too large to fit, or is too large to estimate, break it down. This may mean changing the design of the pro-

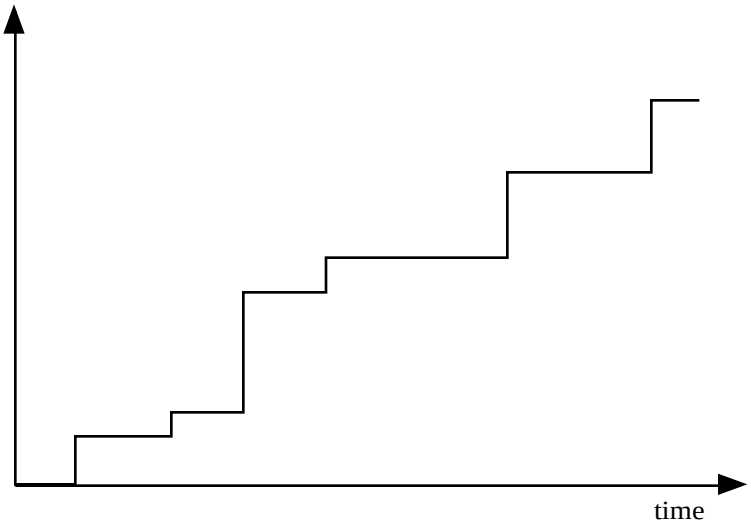



Figure 4. The improvement staircase

duct, or choosing a longer path to eventual full implementation. It's worth doing that to make sure that the product is delivered and to cope with changes of plan. It'll also force you to use flexible and open-ended designs which will allow the project to adapt.[5]

It's only worth planning two or three versions ahead at this level of detail, because the requirements will change and you'll have to do it again at the next version. You should make rough plans further into the future, but detailed plans that far out would be wasted effort.

Specify each version against the critical requirements. This can be as simple as a table showing which attributes you intend to affect and how much (see figure 3).

The task is now to control change to the software to evolve it towards the next version specification, increasing the value of the product. While you're getting there, gather requirements and plan the next cycle, and so on for the lifetime of the product. It's important not to think about starts and finishes, only about continuous evolution and improvement of the product.

## 6. Continuous improvement

This is where we really get into change management. Figure 2 shows how each development cycle takes the product closer to the customer requirements, increasing its value and quality. Figure 3 shows the same thing in a different way, with the master codeline of the product constantly evolving towards the customer requirements.

The key concept of this change management process is *continuous improvement*. Al-

[5] Tom Gilb discusses open-ended designs for management information systems [Gilb88]. The key idea is to choose designs that are easy to change and extend in ways that you haven't anticipated.

ways, always approach the requirements. Never, ever allow a change that takes you further away. This means never allowing a change that dismantles a feature, or breaks a piece of code, even "temporarily". The master codeline must *always* be improving. In practical terms, this means keeping the master codeline ready to build and release to customers at any time, knowing that it will be more valuable to them than before. Figure 4 is a diagram I like to draw to illustrate this simple idea. This is what "software configuration management" is all about [KPA1.1, L2–71].

This makes it easy to meet delivery dates. You can always deliver. The only variation is in exactly what you deliver, and not in when you deliver it. The software from the master codeline is always ready to release. If you've been careful to focus efforts on the critical requirements, to employ flexible design, and to break the work down into small pieces, then you'll always be increasing the value of the product as much as possible by the time of the next delivery.

Most of the rest of this document explains how to achieve this happy state of affairs.

## 7. Issues and changes

Two types of document are used to track and control change: *issues* and *changes*.

An issue is a document that reports that the product doesn't do what the customer wants. This is either because it doesn't meet its specification (a defect) or because the specification doesn't say what the customer wants (an enhancement, or new requirement). There isn't a lot of difference as far as the customer is concerned: the product doesn't do what they want, and they'd like that changed. The process described in this document treats "new development" and "bug fixing" in the same way. Issues are created whenever someone has a problem with the product. Issues subsume bug reports and enhancement requests.

Issues are the only stimulus for change. If change doesn't improve the product for the customer, there's no point in making it.

Issues are examined by management to decide if they're worth pursuing. Defect issues are scheduled depending on their impact. Enhancement issues may result in new requirements (changing the requirements document).

If they're worth pursuing they are analyzed. The problem is carefully understood and reproduced.[6] Various solutions are proposed and maybe prototyped, with estimates of effort required. The issue is then examined again by management, to decide whether to make changes. One or more of the solutions are put into effect by creating changes.

A change document gives instructions to modify the product, and records exactly what was done. It also allows the work to be double-checked to make sure that it actually solves the problem and is a genuine improvement to the product. The change is not allowed into the product until it has been checked. This is achieved by making the change on a branch and only allowing it to be merged after checking (see figure 5).

Change checking is a vital step to maintaining product integrity, and therefore to configuration management. Checking is how the senior engineers can ensure that the

[6] For an enhancement, "reproducing" the problem means simulating the need for the new feature or behaviour.

change fits into the overall design strategy. It's an excellent place to insert a "peer review",[7] a key practice of CMM level 3 [KPA1.1, L3–93]. It's also a stage in which software quality assurance can check that the change complies with the group's best practic-

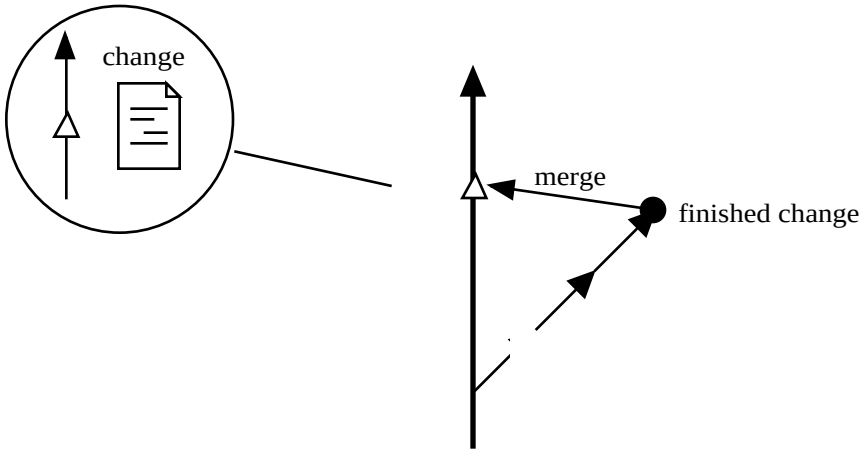



Figure 5. Change branching

es.

At Geodesic Systems, we branch the sources within the Perforce repository as "//info.geodesic.com/change/*N*/...", where *N* is the change document number. One or more people work on the change there, and update the change document as they go. This scheme allows them to work together and submit modifications frequently[8] on their private branch.

## 8. Versions and releases

When a customer finds a defect in the product they want a fix as soon as possible. Technical support policy may be to give the customer a fix or workaround within a few days. The "quick fix" solution to a problem is often not the right solution for the product in the long term. Quick fixes often need to go in without a great deal of thought for consistency with the overall product direction. It's important to separate these fixes from the proper solutions to problems, so that the product doesn't degrade into a pile of hacks.

We can quickly resolve a customer's problem by *patching* the release that they already have. This means making the smallest and quickest change that will fix their problem. This is better than trying to ship them something built from the latest master sources because that may have changed in other ways which will cause the customer problems. Shipping the latest master sources is also risky. They might contain changes that are incompatible with the customer's environment. They also don't have a known specification (so you can't tell the customer what they're getting) and can't easily be maintained later. They probably haven't been thoroughly tested since the last version. Patching a stable re-

[7] I recommend *software inspection* [Gilb95] as a powerful form of review.

[8] I encourage developers to submit every time they make a coherent group of edits, giving them and their colleagues a chance to document the reasoning behind their work along with the edits.

lease gives quicker and higher quality results. This is the main motivation for *version branches* (see figure 6).

A version branch is created by taking a source control branch of the *master sources*

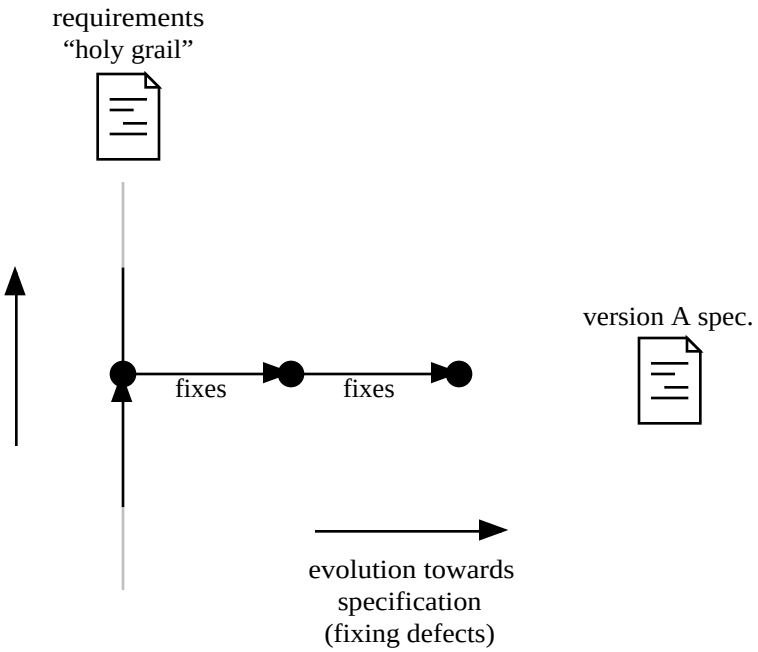



Figure 6. Version branches

when they are believed to meet the version specification. (Or, sometimes, when the deadline has been reached and something has to go out even if it doesn’t quite meet the specification.) The sources on the version branch are called *version sources.* Releases of a version of a product are created from labeled version sources (see figure 6).

Releases of the product are only ever made from the sources on the version branch. Source control labels are used to mark the sources from which a release was made, so that it can be reproduced. In addition, the product image (the thing distributed to the customers) is also archived.

At Geodesic Systems, the master sources, which include the source code, user documentation, design documents, product-related procedures, and test cases, are kept in the Perforce repository at “//info.geodesic.com/*product*/master/...”. The entire tree is branched to “//info.geodesic.com/*product*/version/*version-name*/...” to create a version branch. Releases are built from labels along the version branch. The product image goes in “//info.geodesic.com/*product*/release/*release-name*/...” along with release documentation.

Releases never change. The exact content of a release is committed at the point at which the source control label for that release is created. Any problems with the release must be solved in the *next* release.

Not all releases go to customers. Releases can be created for internal use, as a way of assessing what the quality of the product *would be*. This is especially true of the first release

on the version branch, in which testing sometimes reveals defects that will need to be patched on the version branch.

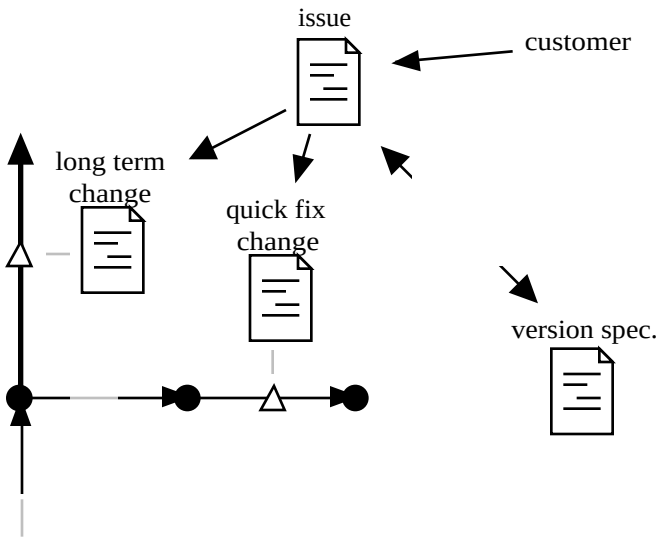



Figure 7. Fixing defects

## 9. Fixing defects

Only changes that fix defects are done on the version branch. A defect is where a release of the product doesn't meet its version's specification. This definition includes most things that are commonly called "bugs",[9] most of which cause the product to fail to carry out one of its functions. In some sense, the version branch evolves towards its specification in the same way that the master sources evolve towards the requirements (see figure 7).

Most importantly, the version sources are a known quantity. Products built from them have been thoroughly tested and released, and so minor changes should produce predictable results. For this reason, it's important that they aren't changed very much, and any fixes done should be as small as possible to solve the specific problem that the customer has. The aim is to get the problem fixed quickly, but without introducing other problems.

Of course, if a defect is found in a version it probably also exists in the master sources as well, and needs to be fixed there too (see figure 6). The fix that is done in the master sources should be a more carefully planned change. Mainly, it needs to be a *maintainable* change, properly documented, and respecting the architecture of the product. This change has to last indefinitely, whereas the change to the version branch only has to last until the customer upgrades to the next version.

This apparent duplication of effort shouldn't be seen as a waste but as an opportunity. Most of the work should be done in the analysis of the issue to work out what should be done. A bandage is then applied to the version sources to fix the problem, while a complete cure is effected to the masters. A "quick fix" might be to describe a workaround to the customer.

## 10. Testing and Tracking

Once a release is built it needs to be tested. The purpose of this testing is to compare the product to its specification so that management can decide whether it's suitable for general release, and to provide feedback for planning as part of "software project tracking and oversight" [KPA1.1, L2–29].

Acceptance testing is driven by version specifications. The version specification says exactly what the product is supposed to be like (unlike the requirements) and the release should match. Acceptance test development can begin as soon as the version is planned, in parallel with product development.

Regression testing is driven by issues. Each issue is covered by a regression test, and possibly more than one. There can be a many-to-many relationship between tests and issues, as long as the relationship is clear. Regression test development can be fed from the issue process, in parallel with solution development.

It's also useful to focus testing on the areas of the software which have changed. Re-

[9] There are two reasons I don't like to use the word "bug". Firstly, "defect" is a better technical term for a discrepancy between product behaviour and specification. Secondly, a "bug" sounds like some sort of external influence on the software, like a cosmic ray, but is usually someone's *mistake* which needs to be understood, corrected, and prevented, not merely "swatted".

gression and acceptance testing can be focused on the areas of recent change by studying the change documents.

The results of testing can be tabulated against the requirements and issues and directly compared with the version plan table (see figure 3), giving a clear idea of the release quality and value. This makes it easy for a senior manager to make an informed decision about making the release generally available.

Defects (differences between the release and its version's specification) are submitted as issues. These can be patched up quickly using the method described in section 9, since we have already created a version branch, and general release is never delayed very long.

## 11. Implementation

It's possible to implement this process in about 12 months in a team of about 20 developers, starting from a situation with no planning or source control.

At Geodesic Systems, I set up an "information server" running FreeBSD, Apache, and p4d, with Apache configured to serve the entire contents of the Perforce repository. I used the fact that Perforce filespecs resemble URLs to make them interchangeable (our depot is called "info.geodesic.com", not "depot"). This makes it extremely easy for people to share information. All the e-mail at the company is archived, with each message at a unique URL. All documents — requirements, plans, versions, releases, issues, changes, designs, procedures, tests, proposals, meeting minutes, and source code files — are stored and indexed in Perforce in HTML so that they can be referenced using URLs. (I especially encourage cross-referencing from code to requirements, designs, changes, issues, and e-mail threads.) The document names are chosen carefully so that cross-references don't break.

The developers use a "handbook" of procedures when working [RB98a], and these procedures are maintained and refined as we learn more about what's effective. We use the version planning, issues, and changes to modify this too. (This is "process change management" [KPA1.1, L5–31], a level 5 practice.)

## 12. Conclusion

In this paper, I've described the best methods I've found for managing software product development, and how these relate to the Capability Maturity Model [CMM1.1].

There's a lot I haven't said about how this method affects the software design and its impact on the working environment and developer attitudes. There are also a lot of small lessons that we've learned and incorporated into our process handbooks. Most of these are specific to our organization and the problems of our software.

The most important thing is that the *organization* is learning by using an information system and a defined software process that can change and evolve just like the software itself. The process, like the product, is never finished until it is dead, and its successors have moved on.

## A. References

[Gilb88] "Principles of Software Engineering Management"; Tom Gilb; Addison-Wesley; 1988; ISBN 0–201–19246–2.

[Gilb95] "Software Inspection"; Tom Gilb, Dorothy Graham; Addison-Wesley; 1995; ISBN 0–201–63181–4.

[RB98a] "Change Management Handbook"; Richard Brooksby; Geodesic Systems; 1998–02–17; <URL: http://info.geodesic.com/proc/cm/>.

[RB98b] "Achieving Quality through Change Management"; Richard Brooksby; Geodesic Systems; 1998–07–03; <URL: http://info.geodesic.com/doc/1998–07–03/quality-change/>.

[CMM1.1] "The Capability Maturity Model For Software, Version 1.1"; Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, Charles V. Weber; Software Engineering Institute, Carnegie Mellon University <URL: http://www.sei.cmu.edu/>; 1993; CMU/SEI–93–TR–24, ESC–TR–93–177.

[KPA1.1] "Key Practices of the Capability Maturity Model, Version 1.1"; Mark C. Paulk, Charles V. Weber, Suzanne M. Garcia, Mary Beth Chrissis, Marilyn Bush; Software Engineering Institute, Carnegie Mellon University; 1993; CMU/SEI–93–TR–25, ESC–TR–93–178.