Requirements Eng (2001) 6:37–50

Requirements Engineering

Preventing Requirement Defects: An Experiment in Process Improvement

Soren Lauesen[a] and Otto Vinter[b]
[a]IT University, Copenhagen, Denmark; [b]Brüel & Kjaer, Naerum, Denmark

Inadequate requirements cause many problems in software products. This paper reports on an experiment to reduce the number of requirement defects. We analysed the present defects in a real-life product and estimated the likely effect of 44 prevention techniques. We had hoped a novel combination of techniques would come up, but the best approach was quite well known, although new to the company: study the user tasks better, make early prototypes of the user interface, and test them for usability. This approach was tried out in a new development project in the same company. Due to the new approach, there was no doubt about requirements during programming, and as a result it became the first project in the company that was completed on time and without stress. Usability was drastically improved, and as a result the product sold twice as many units as similar products, and at twice the unit price.

Keywords: Cost/benefit; Market value; Process improvement; Requirements engineering; Scenarios; Usability

1. Background

The difference between requirement-related defects and other defects in a product is often debated. In this paper we will use this distinction:

- *Implementation defects:* The term *implementation* has many meanings. Here we mean the development activities that produce a workable program. Implementation is mainly carried out by programmers. We have an implementation defect if the product doesn't work as intended by the programmers. Typically, implementation defects show up as program crashes or obviously wrong results.
- *Requirement defects:* We have a requirement defect if the product works as intended by the programmers, but doesn't match the surroundings. One example is that users and customers are not satisfied with it. They may find it too difficult to use, unable to support certain user tasks, etc. Another example is that the program doesn't cooperate properly with existing, surrounding software. Unstated user expectations (tacit requirements), misunderstood requirements and misunderstood existing software are typical causes of requirement defects. The requirement defects can relate to functional as well as non-functional requirements.

Requirement defects may creep in at any stage of development. Many of them creep in at the analysis/elicitation stage, others are caused by developers making wrong guesses during design or programming, and some may even be caused by testing if the tester believes that things should work differently. Implementation defects may also creep in at all stages, except the analysis/elicitation stage. The difference between the two kinds of defects is whether the programmer could see that something was wrong (implementation defect), or whether only users or surrounding systems could reveal the problem.

Defects of both kinds may be detected at various stages of development. The earlier they are detected, the easier they are to repair. Ideally, they should be prevented from creeping in. However, detection as well as prevention requires some effort in addition to usual development. The question is whether it pays to spend this additional effort.

In this paper we discuss only requirement defects. Compared to implementation defects they are more

Correspondence and offprint requests to: Soren Lauesen, IT University, Glentevej 67, DK-2400 Copenhagen NV, Denmark. Email: slauesen@itu.dk

costly to repair. And due to their nature, we need other techniques to prevent or detect them than we need for implementation defects. Programmers can, for instance, find implementation defects through testing or inspection of each others programs, but they can rarely find requirement defects that way. Usually the surroundings must be involved to find these defects.

2. Outline of the Experiment

The purpose of our experiment was to find cost-effective ways to avoid requirement defects in the products. We could see two basically different approaches to this:

1. *The maturity approach:* Compare the existing development processes against one of the maturity models and identify weaknesses. Then improve the weak processes (see Paulk et al. [1]).
2. *The defect analysis approach:* Analyse the defects in present products, identify techniques that could prevent them, and try the best of them in new projects.

The maturity approach is widely used, but we could see no guarantee that the improved processes are cost-effective. Finding the most cost-effective processes was even more elusive. The maturity approach tends to measure its success as conformance to the process model, rather than as improvement on the bottom line. Also, the models have few specific guidelines for better processes.

So we decided to try the defect-driven approach. However, there are few reports on such experiments in industrial settings.

Sutcliffe et al. [2] have investigated a project where they first identified requirement-related defects, weaknesses in the written requirements, and weaknesses in the requirement processes. Next they suggested many improvements to the processes, pointing out that they most likely would have avoided the problems. We see this as a hybrid between the maturity approach and the defect-driven approach. In our project we wanted to narrow down the necessary new processes, estimate their benefits and costs, and try them out in practice.

A straight defect-driven approach is Defect Causal Analysis (DCA; see Card [3]), developed by IBM and used there and at several other places. The principle of DCA is to collect defect reports, find frequent types of defects, discuss them with local developers, and let them suggest improved procedures. Then the new procedures are deployed. Improvements are measured as changes in defect frequencies.

We wanted to do something similar, but had to face several differences. DCA looks primarily at implementation defects, where developers have good expertise. We wanted to look at requirement defects, where developers have less expertise. Typical developers know rather few requirement techniques. Further, they often reject techniques they know, due to the risk involved in any new technique. For this reason we wanted process expert and researcher advice on improvements.

Since our advice came from outside the development team, we reasoned that it was important to motivate developers to use the techniques, and planned to involve them when promising techniques had been identified. An important motivational factor was the expected cost and benefit to their project. We wanted to estimate this before the new techniques were deployed.

DCA is used in large companies (e.g., IBM) and could rely on statistical data to identify benefits over a period of several years. We worked in a smaller company with just 700 employees, 70 of whom were software developers. We had to identify benefits based on a few projects.

We ended up organising the process improvement as a typical action research project. The plan was as follows:

1. Analyse existing requirement defects in a *base project*.
2. Find cost-effective prevention techniques.
3. Use the techniques in a new project.
4. Compare results.

The company was reluctant to spend money on such a project, but the European Union's ESSI programme (European System and Software Initiative) was set up to fund such initiatives. We applied for and got the necessary funding, without which we could not have carried out the experiment.

3. Summary of Findings

The following sections give details of each step. Here is a summary of the findings in each step.

3.1. Analyse Existing Requirement Defects

We classified the requirement defects according to several criteria: error source (where had the true requirement been 'lost'); quality factor (functionality, usability, performance, etc); related interface (user interface, third-party software, etc); cost of handling and repair. The Appendix shows examples of defects and their classification.

The figures showed that about 60% of the defects related to unstated demands (tacit requirements). Almost 70% of the defects had to do with ease of understanding or ease of use (usability). Most defects related to the user interface, but defects with costly repairs related to misunderstood interfaces to third-party software. Most

usability defects had not been repaired because they were not considered 'errors'.

3.2. Find Cost-Effective Prevention Techniques

From the defect analysis, it was tempting to jump to conclusions: since tacit requirements dominate, specify requirements more completely (but how?). Since usability defects dominate, use prototypes plus usability testing or heuristic evaluation (did it pay?). Since problems with third-party interfaces are costly, do something about that (but what?).

Most of these suggestions were rather vague, and did they pay on the bottom line?

Instead of jumping to conclusions, we looked systematically at 44 techniques we knew from literature or found in practice. For each defect we identified the techniques that might find or prevent the defect – and with what probability. On the way, we narrowed down what exactly we meant by using this or that technique. We also invented some new techniques that might prevent defects that seemed hard to prevent in other ways.

It was rather easy to estimate the cost of each technique. However, finding the benefit of preventing a specific defect was difficult. There might be an improved market value of the product or the technique might have implicit effects on other parts of development. We didn't dare to use such benefits as arguments, so we decided on a conservative benefit consisting of the saved costs of handling and repairing the defect. Then it was rather straightforward to find the net cost/benefit of each technique. It was more complex to find the cost/benefit when several techniques were applied together, because each technique filters away some defects, leaving fewer defects for the next technique.

We had imagined that the filtering effect was significant, so that a technique that wasn't very efficient alone could combine in optimal ways with other techniques. This did not occur with our data. In general, the best techniques in combination were the techniques that were also best in isolation.

The conclusion was that only about 10 techniques were worth considering in a project of this kind. The rest were either a waste of time or gave only microscopic net benefits.

3.3. Use the Techniques in a New Project

One thing is to identify promising techniques, another is to introduce them into a new project in the stress of the day. We involved the developers in the final choice. Several surprises came up. (1) Some of our top techniques were useful in one kind of project, but much less important in other projects. (2) The organisational surroundings may block the use of some techniques. (3) Developers have difficulties using many new techniques at the same time. (4) Unforeseen events, such as a new project manager, can overturn earlier decisions to use a certain technique.

However, one project team managed to use two of the promising techniques: (1) bypass marketing and study user tasks directly; (2) make early mock-ups of the user interface and usability test them with real, potential users.

3.4. Compare Results

When the project was completed, we studied the defect reports, the final product, the time spent, and we interviewed developers. We had expected to find small overall performance improvements and a different distribution of defect types. None of this was visible in the figures, and we realised that the projects were so different in their inherent characteristics that these differences were not observable – even if they were there. We had more or less suspected this, and had planned to look for other success indicators.

Among the other success indicators, we found three surprises. (1) The number of usability problems per screen picture was reduced by bout 70% (we had expected 18%). (2) The project was the first one ever in the company that had been completed on time and without stress. The reason was directly attributable to the new approach, where the user interface was designed and usability tested before any part of it was programmed. (3) The product sold twice as many units as comparable products and at twice the unit price. Again the reason was the new approach where user tasks were studied directly and the user interface tested early. The result was a user interface that supported tasks much better and could be used by non-experts. Competitors did not show the same degree of understanding of the tasks and users to be supported.

Compared against our predicted benefits of the techniques, we can conclude that we very much underestimated the secondary benefits. The market value of the approach was big and its influence on other parts of the project highly beneficial. A prediction of this kind early in the project had not been credible, however.

The success of this project – in particular the vastly improved usability – caused other project teams to seek training in the new techniques, and today they are standard approaches in the company. The techniques seem to be equally successful in other projects.

4. The Base Project

The experiment was conducted at Brüel & Kjaer (B&K) in Denmark. They manufacture professional equipment for sound and vibration measurement, and more than half of the product developers are software people. B&K develop products according to a waterfall model where phases can overlap to some extent. They talk about phases such as requirements specification, design, programming, module test, integration test, etc. From the integration test on, B&K routinely records all defects detected by programmers, in-house product testers, marketing and customers. These defect reports were our primary source of data.

B&K had for several years classified the reports according to Beizer's taxonomy [4]. They had successfully improved detection of implementation defects, and now wanted to improve requirements. The experiment was made as a close cooperation between experienced requirements staff at B&K (Vinter) and a requirements researcher (Lauesen).

The first product we studied was a Noise Source Location system (NSL) developed and marketed by B&K. It was a brand-new product that allowed engineers to measure the sound field around an object, for instance a washing machine or an aeroplane, and show the sound field in three dimensions. This is helpful in locating noise sources. As an initial step, the engineer defines a set of grids surrounding the object. Next he measures the sound in each grid point. The results can be shown in many ways: as three-dimensional contour maps, as spectra, as noise power, etc. The system can also control a robot that moves the microphone and makes the measurements.

The system is based on a PC with Windows NT. It connects to various front-end equipment, e.g. a computerised sound measurement unit with calibration and filtering for several microphones. The source code consists of about 90,000 lines of C++, and software development took about 12,000 hours (77 developer months).

The system uses *external software* packages, i.e., packages developed by a third party for a general market. The packages were Windows NT, a 3D-graphics package, and a communication package. It was the first time B&K used these packages in their products. The project team had no influence on the external software packages. If they had defects, the team might report the defects but there was little chance of getting a repair or improvement. Since requirements deal with the relation between the product and all its surroundings, the external software plays a significant role as a potential source of requirement defects.

The requirement specification consists of 20 pages with 107 enumerated and annotated requirements. Figure 1 shows two requirements, R-25 and R-35, chosen to illustrate the style. Note that the specification talks about what the user should be able to see and do, but not how that is to be done. In other words, the user interface (e.g., a prototype) is not part of the specification. Note also that the reason for each requirement is explained in order for developers to better imagine the tasks the user is carrying out. B&K had used this requirement style for some time and were quite satisfied with it.

In addition to this specification, there is a generic requirement specification for all B&K applications running under MS Windows. It consists of 18 pages

Product-specific requirements (examples)
A good way of assuring the measurement quality is to examine the measured spectra. This allows the experienced user to determine the quality of the measurement:

(R-25) During the measurement the application must show the latest measured spectrum.
. . .

It is sometimes impossible to measure some of the desired points. It may be too hot in the environment, or there may not be enough space to position the probe:

(R-35) The application must be able to display all results, even if some of the points have not been measured.
. . .

Generic requirements - company wide (examples)
(G-14) The application must have a graphical user interface compliant with MS-Windows and a common (B&K) style.
. . .
(G-18) Applications must, on user request, be capable of storing their state and later restart in that state.

Fig. 1. Sample requirements from the Noise Source Location system.

with 94 enumerated and annotated requirements. Figure 1 shows two generic requirements (G-14 and G-18).

A supplement to the requirements specification is an object class model on the analysis level. It has served as a cross-check of the requirements.

5. The Requirement Defects

The product had about 800 defect reports when we investigated it a few months after product release. To avoid a heavy analysis burden, we looked at every fourth report – 200 in total – and interviewed developers to find out whether it was a requirement defect or an implementation defect. The distinction was not easy in practice and we grappled for a long time with definitions and classifications. Finally, we used the above definition of requirement defects and then identified 107 reports that related to requirements. We analysed these 107 reports in detail, interviewing developers further as needed.

In this product, slightly more than half of the reported problems were requirement defects. The rest were implementation defects (and a few reports that we couldn't analyse for various reasons).

Beizer [4] reports a much lower fraction of requirement defects. However, we had to include several of his defect classes to cover the above definition of requirement defects. Our figures are similar to those reported by many other companies.

According to the books (e.g., Jones [5]), requirement defects that are detected late should be much more costly to repair than implementation defects. This is probably true, but developers have another choice: reject the problem if it is too costly to repair compared with the perceived benefit. The users just have to live with the inconveniences. In our study we observed few costly repairs, many partial repairs or work-arounds, and many rejections. In general, late in the project, the ideal solution is no longer feasible, so it is important to prevent the requirement defects or detect them early in order to deal with them in a cost-effective manner.

What are the causes of the requirement problems? After many attempts, we used the model shown in Fig. 2. The assumption in the model is that requirement defects are requirements that somehow are 'lost' on their way to the programmers. The model shows how requirements flow to the programmers, and how many requirements are lost or distorted on the way (shown by the arrow thickness and the numbers). As an example, requirements flow from the requirement specification to the programmers. Along this path, 14 requirements in our sample are mistaken. (Actually, there are about 170 written requirements flowing this way, 14 of which are mistaken.)

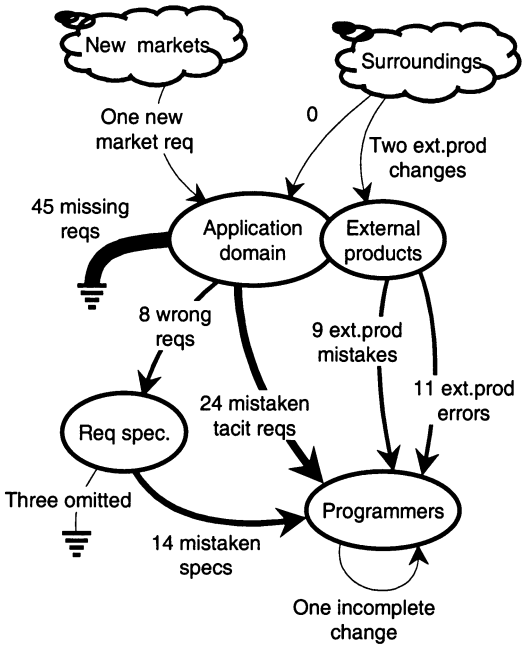

Fig. 2. How requirements pass to programmers or get lost. Arrow thickness and numbers indicate requirements 'lost' on the way. (Based on an analysis of 107 requirement defect reports.)

Some defect reports reflected two 'lost' requirements, so actually there are 118 losses for the 107 reports.

Most requirements start out as demands in the application domain. Some of them become written requirements in the requirements specification, others flow as tacit requirements to the programmers, and some are never transferred to the programmers (missing requirements). Written requirements flow on to the programmers later. During implementation, new demands may flow from new application opportunities (new markets) or the surroundings. Some requirements flow from external products that the system has to cooperate with.

The major source of defects was *missing requirement*s, i.e., requirements that had not been written down in the spec and were not otherwise transferred to developers (45 cases). The cause could be recognised demands that were ignored or forgotten in development (21 of the 45). Or it could be demands that were not recognised although they had been present in the domain all the time (24 of the 45).

The second largest source was *mistaken tacit requirement*s, i.e., the developer somehow knew about the demand, but made a wrong solution (24 cases). The cause could be that the developer made a wrong guess (nine of the 24) or couldn't resolve apparently conflicting or inconsistent demands (15 of the 24).

Note that if product testing had just been a traditional acceptance test, where all requirements are verified, none of these defects had been detected.

Next come *mistaken spec*s, i.e., written requirements that were implemented incorrectly (14 cases). The cause could be a simple mistake (four of the 14), a misunderstanding of the spec (two of the 14), inconsistent requirements (two of the 14) or broad requirements that were not fully implemented everywhere, e.g. that 'the interface shall follow the Windows style guide' (six of the 14).

Defects relating to external software were the hardest to repair or circumvent. In nine cases the developers had misunderstood how external software worked, and in 11 cases the external software didn't work correctly or didn't fulfil expectations.

The diagram shows other, less important, distortions of the requirement flows. In one case a new market was considered for the product in the middle of development. In two cases an external product changed in the middle of development. In eight cases wrong requirements were written down, i.e., requirements that didn't correctly reflect the demand. In three cases a written requirement was forgotten or intentionally left out.

We also classified each defect according to the quality factor that was impaired, e.g., functionality, usability or maintainability. (We used McCall's quality factors [6], which we had good experience with.) Almost 70% of the defects related to usability (ease of understanding and use). We further classified the defect according to the impaired product interface, e.g., user interface, 3-D package, operating system. All the costly repairs that were actually carried out related to the 3-D package and the operating system.

Examples of the different kinds of defects are found in the Appendix. Further statistics of the defects are found in Vinter et al. [7].

6. Prevention Techniques

Although these classifications gave us much insight into the nature of the defects, they were not really useful for finding efficient techniques. For instance, since many defects were caused by tacit requirements, an obvious cure seemed to be a more thorough elicitation of requirements in order to reduce the number of tacit requirements and make them explicit.

However, we could not immediately suggest a few techniques that could elicit most of these tacit requirements. Further, it was difficult to say whether it would pay to use the technique. So a more systematic approach was called for. We decided to look at many potential techniques and systematically identify those defects they could have prevented.

6.1. Potential Techniques

We started out with a list of known techniques ranging from focus groups and usability testing, to inspections and mathematical specifications. As requirement experts we knew many techniques from literature as well as from industry experience. We didn't care whether the techniques had tool support, full manuals, etc. That was a matter of cost to be considered later. Our prime concern was whether the technique might be able to find or prevent some of the defects we had observed.

While we classified the defects, we tried to imagine what could have prevented each defect. When no technique on the list seemed able to prevent the defect in question, we tried to invent a new technique or a variation of an existing one. As an example, we knew no technique that could have revealed some hard problems with the 3-D package (defects D1 and D389 in the Appendix). So we invented three techniques:

1. *External software modelling:* Model the third-party package as objects and operations, and test that the model is correct.
2. *External software stress test:* Test the third-party package against the planned product design with realistic data in extreme cases. (A kind of prototype exercising the package.)
3. *External expert review:* Have a package expert review the planned product design to point out potential problems with using the package in that way.

We removed many well-known techniques from the final list, because we could see no use for them in relation to the actual defects. Initially, for instance, we thought that argument-based techniques could be useful, e.g., gIBIS [8]. They might be useful in other projects or during design, but our defect reports did not show a need for them.

The net result was a list of 44 promising requirement techniques, divided into these eight groups:

1xx: Elicitation techniques: focus groups, scenarios, work with users, etc.
2xx: Usability techniques, prototype testing, style guides, heuristic evaluation with domain experts.
3xx: Checks of third-party packages.
4xx: Tracing requirements to design documents.
5xx: Risk analysis.
6xx: Formal specifications.
7xx: Inspections, checks, and reviews.
8xx: Specify non-functional requirements better: performance, robustness, interoperability, etc.

As an example, technique 230 (in group 2xx, subgroup 23x) prescribes 'usability test of a functional prototype with daily tasks'. A full description of all the techniques is found in Vinter et al. [7].

6.2. Estimating Hit-Rates

The next step would be to find the best of these techniques. However, what was 'best'? The techniques that best improved the market value of the product; or the techniques that best reduced the development cost? We will discuss that later, but as an initial step we wanted to see which defects each technique could prevent.

To do this, we estimated the hit-rates for each combination of defect and prevention technique. The hit-rate $h(t,d)$ is the probability that technique t will prevent defect d. Preventing a defect means that the technique either has to prevent the defect, or at least detect the defect sufficiently early so that the defect report wouldn't be made.

We were three experts looking at each defect to identify the techniques that might have revealed the problem, and estimate their hit-rates. To avoid hair splitting, we decided to use only hit-rates of 0, 5, 20, 50, 80, and 95%. (These figures were our numeric version of a subjective scale like *impossible*, *unlikely*, ... *almost sure*.)

We didn't take averages, but reached consensus for each hit-rate. When we disagreed, we explained how we imagined that the technique would be carried out. Usually we realised that we disagreed on the hit-rate only because we had different assumptions on how well developers would master the technique, how carefully the technique was applied, and how well the result was checked. We then agreed on how to carry out the technique, improved the description of the technique accordingly, and had no difficulty stating the hit-rate in question. As an example, technique 230 (usability test of functional prototype with daily tasks) was defined as a particular variant of usability testing when our discussions revealed different assumptions on the kind of prototype and which user tasks to use for testing.

For each defect, there were around five prevention techniques with a positive hit-rate. Unfortunately, around three of them had a low potential as we estimated that they would hit the defect with at most 20% chance.

About 25% of the defects were hard to prevent by any technique. Even the best of our techniques would hit them with at most 20% chance. Almost 40% of the defects could be easily prevented, because some technique would hit them with at least 80% chance.

When we had estimated all hit-rates $h(t,d)$, it was easy to find the total hit-rate for each technique as the sum of $h(t,d)$ for all d. This gives the expected number of defect reports that would have been prevented in the sample. Scaling up from the sample to the entire set of error reports gave the expected number of defect reports that would have been prevented in the entire project. As an example, we have these figures for the apparently strongest of our techniques, technique 230:

> *Usability test of functional prototype with daily tasks (technique 230):*
> Sum of hit-rates, h(230,d): 22.6 out of 200 defects
> Expected defects found in total project: 90.4 out of 800 defects

In other words, in projects like NSL, we would expect technique 230 to prevent an average of 90 defect reports, i.e., 11% of all reported defects or 20% of the requirement defects.

The weakness in this approach is, of course, that the figures depend entirely on the expert's ability to estimate the hit-rates. We have tried to guard against this by using three experts with different backgrounds and insisting on them reaching consensus rather than taking an average.

Later experience showed that we grossly underestimated the hit-rate of usability tests – or maybe there was an accelerator effect at work, so that removal of some usability defects caused users to pass over other potential defects (see Section 8.2).

In another case, we hit quite precisely. We had estimated that a careful inspection of the requirements specification would hit an average of two defects in our sample – a bit low, considering that inspection is considered a very important technique. Later we remembered that such an inspection *had* been made. Looking at the inspection report, we found predictions of exactly two of the defects in our sample. Counting as above, we would expect that the report predicted a total of eight defects in the entire project.

Actually the inspection report contained 84 predictions of wrong, missing or ambiguous requirements. But most of them were 'false positives' in the sense that they predicted problems that did not appear in the defect reports. Although the requirements looked wrong or ambiguous, the developers had got the necessary tacit understanding to avoid creating defects.

6.3. Cost of the Techniques

Based on the detailed technique descriptions, it was rather straightforward to estimate the cost of each technique, measured as the number of work hours needed to carry it out. We assumed that the technique

had adequate tool support. With the rather short requirement specs we dealt with, it wasn't a critical issue anyway.

As an example, we had this cost calculation for the usability test technique above (training time for developers were included but have been omitted here for simplicity):

Usability test of functional prototype with daily tasks (technique 230):

Cost estimate	*Hours*
Test planning, designing about 30 tasks	30
Basic functionality for 35 windows, one day each	245
Three test sessions, each with 3 users	54
Total	329

This calculation covers only the time to detect the usability defects. The time to repair the defects is not considered, as we argued that it would occur also if the defects were detected late and then corrected.

6.4. Benefit of the Techniques

Estimating the benefit of each technique turned out to be very hard. Ideally, we wanted to look for improved market value of the product, but we couldn't find reliable ways to assess it. The literature didn't give us many clues to the real-life benefit of each technique. After some time, we realised that many benefit factors were involved:

1. Saved work hours to report and handle a defect.
2. For reported defects where some repair was made: saved work hours for repairing the wrong solution (net rework time).
3. For reported defects that were not repaired or repaired only partially: the market value minus development time if it had been dealt with earlier.
4. Effect on early development if requirements had been better known.

Factors 1 and 2 dealt with simple savings in development time, and in principle they could easily be estimated. Surprisingly, most defects were either considered unnecessary to repair, or just a few hours were spent repairing them. In many cases the repair was only partial or a work-around, because the kind of solution one would have made if realising the problem early was too costly at the late stage. An example is given for defect D1 in the Appendix.

We realised that factors 3 and 4 might be very important, but it was impossible to give a reasonable estimate of them. To avoid being accused of unrealistic expectations, we included only the tangible benefits of factors 1 and 2 in our cost/benefit calculations. Actually, factors 3 and 4 turned out to be highly significant, but we couldn't have estimated their effect early.

The pragmatic decision was a prevention benefit for each defect of either 5, 25 or 50 hours. Surprisingly, 100 out of the 107 defects would save just 5 hours each if prevented. The total expected benefit if all requirement defects were prevented was as follows:

Total possible benefit in NSL project:

Prevention benefit of	5 hours:	100 defects
Prevention benefit of	25 hours:	2 defects
Prevention benefit of	50 hours:	5 defects
Total saved in sample:	800 hours:	107 defects
Total saved in project:	3200 hours:	428 defects

Compared to the total development cost of 12,000 hours, this saving is about 27%. Unfortunately this does not include the cost of the necessary prevention techniques. Carrying out all of the 44 techniques would cost about 6000 hours and still we would catch less than 60% of the defects. The result would look real bad on the bottom line.

In order to find the best techniques, we computed the net benefit of each technique. As an example, we can compute the net benefit for the usability test technique above as follows: it had a 5% hit-rate for eight defects, each of which would save 5 hours if prevented. These defects would save an average of 2 hours total. It had a 20% hit-rate for 23 other reports, etc. In total the computation looks like this:

Technique 230 (Usability test with . . .)				*Hours saved*
8 reports with	5% hit-rate and	5 hours	benefit:	2
23 reports with	20% hit-rate and	5 hours	benefit:	23
14 reports with	50% hit-rate and	5 hours	benefit:	35
7 reports with	80% hit-rate and	5 hours	benefit:	28
4 reports with	95% hit-rate and	5 hours	benefit:	19
2 reports with	20% hit-rate and	50 hours	benefit:	20
1 report with	80% hit-rate and	50 hours	benefit:	40
Total benefit in sample (hours)				167
Total benefit in project (hours)				668
Cost of technique (hours)				329
Net benefit (hours)				339

According to the calculation, the total benefit for this technique would be 668 hours. Carrying out the technique, however, would cost 329 hours, so the net benefit would be just 339 hours, or about 3% of the entire development time.

6.5. Selecting Techniques

Calculations showed that more than half of the techniques would be a waste of time. They had higher costs than benefits according to our estimates, so the best

combination of techniques was to be found among the rest. When using techniques in combination, the benefits are reduced because each technique filters away some defects, leaving fewer defects for the next technique. Taking this into consideration, we could compute optimal combinations of techniques by means of dynamic programming.

According to our calculations, the best combination of any four techniques would prevent 37% of the requirement defects and reduce the total development cost by 6%. Combinations with more than four techniques showed very little improvement.

We observed that the filtering effect had little influence on the optimal combination of techniques. In general, the best techniques in combination were also the best in isolation. For instance, the six techniques that formed the best combination were among the top eight techniques in isolation. Further, the two top eight techniques left out were variants of one of those included.

We had hoped that somehow a few techniques farther down the list could combine like a jigsaw puzzle and detect most defects. This was not possible according to our calculations.

The conclusion is that we could select the best techniques one by one rather than as an optimal combination. This was very important when we came down to selecting the techniques in cooperation with developers. It allowed us to consider organisational factors in addition to the cost/benefit estimate, without invalidating particular optimal combinations.

We tried a few variations of the benefit factor. Above we used a simple net benefit (benefit minus cost) as the important factor, and we also tried benefit divided by cost to optimise the return of the hours invested in the techniques. The results didn't change a lot.

However, the calculations showed that the optimal combinations were rather sensitive to the assessment of individual, expensive defects. As an example, we had lengthy discussions about the hit-rate for the last defect in the above table. Much later we realised that a change from 80% to 5% for this single defect would have caused the entire technique to move from number two on the top eight list to number six. In contrast, the sensitivity for the 'cheap' defects (the majority) was small.

Irrespective of the calculations, we were convinced that one technique was highly important: studying potential user tasks and writing down the results as *scenarios* (technique 101). There were two reasons for it: (1) The technique would be useful for identifying the tasks to be used in usability tests. (2) We had seen in another project that market opportunities could be improved this way. The choice of this technique could not be justified by the NSL defect reports, because they focused on defects in the product, not on lost market opportunities. As we will see later, this technique became highly important.

Scenarios mean different things to different people, as explained by Campbell [9]. In our variant, we study what the users actually do or try to do – irrespective of any possible computer support. We write down the results as descriptions of the user profile, the work environment and the user tasks. Our approach is much like the ones reported in Carlshamre and Karlsson [10], Graham [11], and Hooper [12]. We do not mean scenarios in the object-oriented or UML sense, which is quite close to the computer and describes the detailed interaction with a proposed computer solution.

The end result was that we presented two development teams with a list of potentially good techniques, including the following:

101 Scenarios, including description of user tasks.
230 Usability test of daily tasks with a functional prototype.
220 Usability test of daily tasks based on a screen mock-up.
280 Product expert screen review (a kind of heuristic evaluation).
301 External software stress test. Aimed at preventing the very costly defects associated with new external software.
721 Orthogonality check of object model. A specific check to see whether an operation (or feature) could be applied to all user objects where it might be useful.
730 Initial value check. A specific check to see whether screens would contain the proper values initially.
820 Performance specification. A check to see whether proper limits had been defined for certain performance factors (time, size, amount, and precision).

The developers selected five techniques based on many criteria: e.g., the cost/benefit that we had calculated; whether the technique matched their project; whether there were non-quantified advantages and disadvantages of the technique; whether the technique seemed to duplicate other techniques.

During these discussions we realised that the best techniques in NSL might not be the best in other projects. As an example, the external software stress test, which had been the top technique for NSL, was useless in their projects since they didn't use new external software.

Usability test with a functional prototype had been top two in our calculations, and the teams favoured it. But it had a serious disadvantage: the rather large effort spent on making a functional prototype meant that the team would be reluctant to discard the prototype and rethink the user interface completely. We therefore persuaded the teams to replace it with the technique 'usability test

based on a screen mock-up'. This too turned out to be an important decision.

The teams received training in the five techniques they had selected and started using them in the project. Shortly after, they realised that it was too many new things at once. It seems unrealistic to introduce more than one or two techniques at once in the midst of all the other daily activities. (Card [3] has made similar observations for Defect Causal Analysis). So actually they used only two new techniques:

1. Scenarios, including description of user tasks (technique 101).
2. Usability tests with daily user tasks, based on screen mock-ups (technique 220). This technique used prototypes with carefully designed screens and some screen switching features, but no real functionality.

According to our estimates, these two techniques should be able to prevent about 15% of all requirement-related defects. They should save about 3% of total development time (350 developer hours).

Many developers believe that usability testing is very costly, but we had good experience using a low-cost usability test, and that was what we prescribed (see Jørgensen [13] or Lauesen [14]). For a more full description of usability testing, see Dumas and Redish [15].

Since both techniques are rather usability-oriented, you would expect that they mainly detect usability defects. This wasn't the case. Calculations showed that they should detect 18% of all usability defects (13 out of 72 usability defects in our sample) and 11% of the others (functionality, interoperability, etc.).

7. Using the Techniques

Both teams used the techniques with great enthusiasm. However, there was a major barrier to overcome: getting access to real users (customers). In a product-developing company like B&K, marketing is the link between developers and customers, but marketing staff are reluctant to give developers direct access to customers. Also, in an international market, it is hard to find representative users. These problems are known at many companies as described by Grudin [16]. In spite of all, developers managed to get across the barrier.

Team A developed a new product, a portable sound intensity meter (PT-2). The hardware was available: a nice, slim case with a special screen and keyboard, CPU, exchangeable flash memory, battery, two microphones; length 50 cm, total weight 3 kg. The product reused the operating system and some other basic modules from a different portable product.

Developers expected it to be used when measuring noise intensity in houses, traffic noise, etc. Anyone could imagine how these tasks were performed, but just to make sure they decided to observe potential users and write scenarios and task descriptions.

To their surprise a major demand was to measure noise levels on the surface of tall ventilation shafts, chimneys, etc. One scenario looked like this (much abbreviated):

> *Chimney scenario.* The user climbs the chimney on the small steps attached to it, carrying the meter. With an arm stretched out around the chimney, he measures the noise level in various points. He checks that the measurements are of adequate quality, repeats measurements as needed, and climbs back down.

This revealed a few problems in the planned design. (1) Although portable, there was no easy way to carry the sound meter on the ladder. (2) The user had to answer a few dialogue boxes to start the measurement (with arm stretched out). (3) Reviewing the measurements for quality was not easy standing on the ladder because the display was upside down relative to normal operation.

The team came up with a revised design of the meter. It should have a carrying belt. The user dialogue should allow single-button start and stop of measurements with audible feedback. The display should be reversible programmatically, etc.

These details made the meter a great success when released. Competitors did not show the same understanding of the tasks to be supported.

The team made usability tests very early, testing the chimney scenario among other scenarios. The tests made them revise their first design completely, and the next design significantly. Had the team developed functional prototypes, the cost of doing so would have discouraged them from significant redesigns. The mock-ups, however, could be redesigned in a day or two.

This approach made the product so easy to use that even non-experts could use it.

What did the requirement specification look like with the new approach compared to the old approach? In principle the scenario descriptions and the prototypes could largely replace the traditional numbered requirements. (Some numbered requirements had to remain, for instance platform requirements). But this step was not taken. The old kind of numbered requirements remained, but scenarios and prototypes were supplementary information serving as justification of the requirements and as well-tested examples of how they could be implemented.

Team B developed another new product, a sound power meter. They followed a similar approach, wrote scenarios, produced and tested prototypes. But then something unexpected happened. The project got a new project manager. He didn't believe in the techniques,

discarded the prototypes, and designed the user interface in a style he knew from another, typical B&K product. Development then continued from there. The team overshot their budget significantly and didn't complete on time. However, we have not had opportunities for studying the project further.

8. Results

When team A had completed the project, we started analysing the effects of the new approach. Could we see a reduction of 15% defect reports? Could we see a reduction in work hours? Well, we hadn't imagined how difficult it was to compare two very different projects. Traditionally one would compare the number of defects per KLOC (thousand lines of code), but it didn't make much sense in our case because one project struggled heavily with new external software, while the other largely reused special platform software they knew very well, but made comparatively many screen pictures.

To improve our understanding of the results, we looked at the previous product based on team A's portable platform, and analysed its defect reports. So in total we had three projects to compare:

1. *NSL:* The first product we studied (the *base projec*t). It was the first one in B&K that used Windows NT. It also used 3-D presentations of measurements for the first time.
2. *PT-1:* The previous product on the portable platform. It was developed by some team A members plus other staff. The main focus in this product had been the technical side of this kind of sound measurement.
3. *PT-2:* The new, different product on the portable platform, developed by team A. Nobody in the team had been involved with NSL. The main focus in PT-2 was on the usability side. The complexity of the screen pictures was comparable to the complexity of the windows in NSL.

Table 1. Comparison of old projects (NSL and PT-1) with PT-2 that uses the new approach

	NSL	PT-1	PT-2
Developer months	77	20	19
New screen pictures	45	6	23
Implementation defect reports	352	113	71
Total req. defect reports	428	77	66
Usability defect reports	288	41	43
Total req. defects/month	5.6	3.8	3.5
Usability defects/month	3.7	2.0	2.3
Non-usability req. defects/month	1.8	1.8	**1.2**
Implementation defects/month	4.6	5.6	**3.7**
Usability defects/screen	6.4	6.8	**1.9**

In Table 1 are some figures for comparing the three projects.

8.1. Looking for Planned Benefits

One goal was to reduce development time, but it is hard to tell whether total development time was reduced. One reason is that an expected reduction of 3% is hardly possible to measure, given all the other differences between the projects – particularly in a company with little systematic recording of employee time. PT-1 used about the same number of developer months as PT-2. Although the team produced four times as many screens, they didn't grapple with new measurement techniques.

Did we reduce the number of requirement defects? The total number of requirement defects per month is much smaller than in NSL, but roughly the same as in PT-1. So there doesn't seem to be a significant reduction from PT-1 to PT-2. However, this may be a result of two opposing changes, shown in the table: slightly more usability defects per month in PT-2 (due to many more screen pictures, which still carry a relatively high defect rate), and fewer requirement defects of other kinds. Fewer requirement defects of other kinds are very likely, given that much measurement software was reused.

8.2. Unexpected Results

We had suspected that it might be a problem to compare the projects in the planned way, because they were rather different. So we looked at other indicators too.

The most striking effect was that the new techniques had reduced the number of usability problems per screen by about 70%. This was most likely the result of usability testing and willingness to redesign the interface. However, the effect was much more than the 18% reduction in usability defects that we expected in our expert predictions. It was even more surprising when you knew that the usability tests were carried out far less carefully than prescribed in the original description of the technique. Either the experts were too pessimistic or some accelerator effect was at work.

The accelerator effect could work as follows. Assume that we have a user interface with no usability problems in the central screen pictures. It might then suggest a 'correct' mental model to the users that would allow them to better understand other less frequently used screens. In other words, removal of the central usability problems would make the users pass over other potential usability problems. Since the usability test covered only the most central screens, this could explain its much larger effect.

A small effect is that the number of implementation defects per month is reduced by more than 20%. This was unexpected – why would better requirement techniques reduce raw programming errors? An independent researcher, Jan Pries-Heje, had interviewed developers during the project, and his report provided the answer:

According to the developers, the new approach created a solid foundation at an early stage. There was no doubt about requirements and user interface during the rest of the development. Detailed design and programming became straightforward tasks. Previously, things had to be changed all the way during programming and integration. With the new approach, there could still be new requirements coming in from marketing, but developers responded by asking for a scenario where this feature would be useful. Only rarely could such a scenario be identified.

Developers had also explained that development proceeded without the usual stress and still on time. Further, already the first trade fair showed high customer satisfaction. Competitors did not show the same understanding of the customer's needs.

Hearing about these results, other project managers wanted to learn the new techniques, and several courses had to be given. These project managers found the techniques effective too, and today the techniques are used throughout the company.

Later, sales figures showed that the PT-2 product sells twice as many units as comparable B&K products, and at twice the unit price.

9. Discussion

Looking at the final choice of techniques (scenarios and early usability testing), you may wonder why it took so much effort to reach a conclusion that afterwards looks obvious. Did we just reinvent the wheel? No, we knew that the techniques existed, but little was known about their effectiveness in a real-world project. We managed to estimate the effectiveness. We also learned a lot about how to select and introduce new techniques in general, which is a critical point in action research.

Scenarios and usability testing are not widely used, but Weidenhaupt et al. [17] identified and compared 15 European projects that used some kind of scenarios. There was a difference in the approaches, but the typical approach was very similar to the one B&K chose: scenarios are initially used to describe what is going on in the user domain without regard to the exact role of the new product. Later, developers make prototypes and usability test them against the scenarios.

However, it was not possible to quantify the advantages of the techniques in those projects.

The B&K projects seem to differ from those 15 projects because B&K develop products with embedded software for a market. We believe this difference helped us quantify the effectiveness of the techniques because a lot of the environmental factors, e.g. the development process, market analysis and defect reporting, were the same from one project to another.

Still the B&K projects were so different technically that we could not see improvement in developer performance such as defects per person-month or KLOC per person-month. We could see other significant effects, however.

10. Conclusion

We can summarise our conclusions in this way:

1. Scenarios and early usability testing (based on a mock-up) are highly beneficial techniques. They gave a much more predictable development, had a low cost, reduced the number of usability defects by 70%, and vastly improved user satisfaction as shown by market acceptance and sales price.
2. Looking at the individual requirement defects, we were able to identify or invent cost-effective prevention techniques. In contrast, classifying the defects according to the source of the defect, the kind of requirement violated, etc. did not help identify or invent cost-effective prevention techniques.
3. Selecting the best techniques involves many factors, some of them quantifiable, others more subjective. Estimating the hit-rate of a technique is helpful, but is only one of the factors.
4. Estimating the value of a prevention technique is possible if you look only at the saved 'rework' time. For some techniques, however, there are huge secondary benefits in improved market value and improved overall development, as observed for our chosen techniques. For other techniques such secondary benefits are dubious.
5. A technique may be very beneficial although it cannot be justified by defect prevention. The use of scenarios is one example. It vastly increased market share, although it hardly prevented any defect reports. Lost market opportunities are simply not recorded as 'defects'.
6. Tangible benefits (e.g., the number of prevented defects) are important to convince developers to use a new technique. However, the value of a technique depends on the kind of project. What is a definite advantage in one project may be a waste of time in another.
7. One could imagine that a combination of medium-hit

techniques would be advantageous to a single high-hit technique. According to our calculations, this did not happen in our case. Techniques that were good in combination were also good in isolation.

8. Developers cannot handle more than one or two new techniques at a time. This is no problem according to our calculations. If you first introduce the best single technique, then the next best and so on, you end up with an optimal combination of techniques. There are probably exceptions to this, but none we could see in our case.

Acknowledgements. The project was funded by the European Union's ESSI programme (European System and Software Initiative) under the name PRIDE (a methodology for Preventing Requirements Issues from becoming Defects, Project no. 21167). Without external funding, B&K would never have commenced such a project. The authors would like to thank Per-Michael Poulsen (the third expert), Kai Ormstrup Jensen and Jan Pries-Heje for their work and dedication while analysing reports, studying the development process and helping with ideas.

References

1. Paulk MC, Curtis B, Chrissis MB, Weber CV. Capability maturity model for software, version 1.1. 93-TR-024, Software Engineering Institute, Pittsburgh, PA, 1993
2. Sutcliffe AG, Economou A, Markis P. Tracing requirements errors to problems in the requirements engineering process. Requirements Eng 1999;4:134–151
3. Card DN. Learning from our mistakes with defect causal analysis. IEEE Software 1998;January/February:56–63
4. Beizer B. Software testing techniques (2nd edn). Van Nostrand Reinhold, 1990
5. Jones C. Applied software measurement. McGraw-Hill, New York, 1991
6. McCall JA, Matsumoto M. Software quality metrics enhancements, Vols I–II. Rome Air Development Centre, 1980
7. Vinter O, Lauesen S, Pries-Heje J. PRIDE final report: a methodology for preventing requirements issues from becoming defects (PRIDE). ESSI project no. 21167. Brüel & Kjaer Sound and Vibration Measurement, May 1999. Also at www.esi.es/VASIE/Reports/All/21167
8. Conklin J, Begeman, M. gIBIS: a hypertext tool for exploratory policy discussions. ACM Trans Office Inf Syst 1988;6(4): 303–331
9. Campbell RL. Will the real scenario please stand up? SIGCHI Bull 1992;April:6–8
10. Carlshamre P, Karlsson J. A usability-oriented approach to requirements engineering. In: Proceedings of ICRE'96. IEEE Computer Society Press, Los Alamitos, CA, 1996, pp 145–152
11. Graham I. Requirements engineering and rapid development. Addison-Wesley, Reading, MA, 1998
12. Hooper JW, Hsia P. Scenario-based prototyping for requirements identification. ACM SIGSOFT, Software Engineering Notes 1982;7(5):88–93
13. Jørgensen AH. Thinking-aloud in user interface design: a method promoting cognitive ergonomics. Ergonomics 1990;33(4):501–507
14. Lauesen S. Usability engineering in industrial practice. In: Howard et al. (eds). Human– computer interaction (Interact'97). Chapman & Hall, London, 1997, pp 15–22
15. Dumas JS, Redish JC. A practical guide to usability testing. Ablex, Norwood, NJ, 1993
16. Grudin J. Systematic sources of suboptimal interface design in large product development organizations. Human–Computer Interaction 1991;6:147–196
17. Weidenhaupt K, Pohl K, Jarke M, Haumer P. Scenarios in system development: current practice. IEEE Software 1998;March/April:34–45

Appendix: Examples of Defect Reports

Below follow examples of defect reports from NSL. There is at least one example for each of the main sources of defects. In the examples we refer to various quality factors according to McCall's classification.

The defect reports in the original files were numbered from one upwards. Below, D461 etc. refer to these original numbers. The explanations are not the original ones – which would be impossible for outsiders to understand – but attempts at more popular explanations. Still, we have selected examples that are easier to understand than the average. All defects below were easy to repair, or at least work around, except for defect 389.

New market requirement

D461: The product should show a PI index in order to determine sampling time. A PI index is convenient for setting up robot control and knowing manual sampling time. This is related to a potential new application area, where the microphones were robot-controlled. From a quality factor viewpoint, it was a question of missing functionality.

External product changes

D441: The communication format in another, independent B&K product changed in a new release. This product was used as a front-end to NSL and provided the measurement data. The NSL product crashed when communicating with this new version. From a quality factor viewpoint, it was a question of robustness, interoperability and maintainability.

Missing requirement and external product mistake

D1: The essential feature of the NSL product was to show spectra, etc. on a 3-D surface. An external (third-party) graphics package was used for the 3-D display. It turned out that when a 3-D picture was rotated or zoomed, the annotations on coordinate axes, grid surfaces, etc. were also rotated and zoomed. This could make the annotations unreadable.

The problem related to both the user interface, where it was a missing requirement, and to the external

software interface, where it was a mistaken specification of external software. From a quality factor viewpoint, it was a question of usability. It would have been very expensive to repair the defect at this stage, so only a work-around was made in the form of a separate annotation window.

Wrong requirement spec

D133: When measurements in a grid are missing, the product should interpolate a value according to requirement R-35. However, users were confused whether the point had been measured or not, so the requirement was (partially?) wrong. From a quality factor viewpoint, it was a question of usability.

Mistaken tacit requirement

D137: The product could measure sound in a grid of points. The measurement sequence could be Z-shape, S-shape, or Free shape. However, users couldn't find a 'Free shape' button. Two buttons said Z-shape and S-shape. When both buttons were out, it meant Free-shape, which the users didn't figure out. From a quality factor viewpoint, it was a question of usability. Although the developer understood the tacit requirement for choosing between the alternatives, he made a wrong guess on how to deal with it.

External product error

D389: The external 3D package couldn't correctly hide surfaces behind narrow objects. A small gap was needed to tell the package that a grid inserted into a larger grid should hide the main grid. From a quality factor viewpoint, it was a question of correctness in the external software. A real repair was needed, amounting to an estimated 50 work hours.

Omitted requirement

D189: According to the written requirements, 'zero interpolation' should be available in grid maps. The feature was omitted without changing the requirements because the external package didn't support it. Old B&K systems supported it, but the facility was not important.

The problem related to both the user interface, where it was an omitted requirement, and to the external product interface, where it was an error in the external software. From a quality factor viewpoint, it was a question of usability.

Mistaken requirement spec

D501: When the user clicked on the scroll bar in the Map Table (outside the arrows), the window scrolled one line only, rather than a page. Although the developer understood the broad requirement of following the Windows style, he made a mistake in this case. From a quality factor viewpoint, it was a question of usability.

Incomplete change

D773: In some cases a large text was clipped so that it became unreadable. It was a consequence of repairing other requirement defects through computing the text size, rather than having a fixed size. In this case, however, the size was computed wrongly. Prevention of some other requirement defects (D737 and D1) would have prevented this too. From a quality factor viewpoint, it was a question of correctness.