From Proceedings of the Sixth International Workshop on Requirements, Stockholm, REFSQ'2000.

Preventing Requirement Defects

Søren Lauesen & Otto Vinter

IT University, Glentevej 67
DK-2400 Copenhagen NV
slauesen@itu.dk

Brüel & Kjaer
DK-2850 Naerum
vinter@inet.uni2.dk

Abstract. Inadequate requirements cause many problems in software products. This paper is a case study of requirement defects in a real-life product. We analyzed the cause of the defects and estimated the likely effect of about 50 prevention techniques. We had hoped a novel combination of techniques would come up, but facts suggested that the best approach was quite traditional, 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. Since the two projects differed in many ways, it was harder than expected to compare development costs and defect statistics. However, two unexpected results turned up: We could observe a drastic decrease in the number of usability problems per screen. Further, the new project was the first and only one among scores of other projects that was completed on time, without stress, and with high customer satisfaction. The reason, the developers reported, was that due to the new approach, there was no doubt about requirements during detailed design and programming. They became straightforward tasks. The new approach spread rapidly and is now applied on a regular basis in the company.

1. Background

Problems and defects in software products fall into two main categories:

Implementation defects: The product doesn't work as intended by the developers. As an example, the program crashes or produces obviously wrong results. Program bugs are the typical cause of this; for instance that the developer made a mistake or thought that other pieces of the program worked differently.

Requirement defects: Although the product works as intended by the developers, the users and customers are not satisfied with it. They may find it too difficult to use, unable to support certain user tasks, etc. Unstated user expectations (tacit requirements) and misunderstood requirements are typical examples.

The purpose of our case study was to find the causes of requirement defects, the kind of consequences they had, and efficient prevention techniques.

The study was made 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. They had successfully improved detection of implementation defects, and now wanted to improve requirements. The case study 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 allows engineers to measure the sound field around an object, for instance a washing machine or an airplane, and shows 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 computerized 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 there were 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. Here are two requirements, R-25 and R-35, chosen to illustrate the style:

> 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.

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. as a prototype, is not part of the specification.

There is also a generic requirement specification for all B&K applications running under Windows. It consists of 18 pages with 94 enumerated requirements. A supplement to the requirement specification is an object class model on the analysis level. It has served as a cross check of the requirements.

2. Requirement Defects

Brüel & Kjaer had practiced systematic error reporting for several years and routinely classified the reports according to Beizer's taxonomy [Beizer, 1990]. The product we studied 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 figure out whether it was a requirement or an implementation defect. The distinction was not easy in practice and we grabbled 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. The remaining 93 reports were implementation defects and a few reports that we couldn't analyze for various reasons.

We analyzed the 107 requirement reports in detail, interviewing developers further as needed.

It turned out that about 65 of the reports dealt with *tacit requirements*, i.e. requirements that had not been written down. The developers had been aware of some of them, but failed to implement them for some reason. Other tacit requirements were a surprise to developers.

About 20 reports dealt with stated requirements that were wrong, misunderstood, forgotten, etc. Three reports dealt with new requirements caused by changes in the surroundings.

The remaining about 20 reports dealt with defects in external software packages or misunderstandings of how they worked. These defects were the most costly to repair or circumvent.

Here are some examples of reported requirement defects:

Tacit, surprising requirement

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

It would have been very expensive to repair the defect at the late stage where it was detected, 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.

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. A real repair was needed, amounting to an estimated 50 work hours.

3. Preventing the defects

An obvious cure to many of the problems would seem to be a more thorough elicitation of requirements in order to reduce the number of tacit requirements and make them explicit. However, it turned out to be difficult to come up with a few techniques that could elicit most of these tacit requirements. Further, most techniques seemed able to prevent several kinds of observed defects to some degree. 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.

3.1. Potential Techniques

We started out with a list of known techniques ranging from focus groups and usability testing, to formal specifications and inspections. As requirement experts we knew many techniques from literature as well as 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. The initial list was about 50 techniques.

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. Later, we improved and specified each technique further. The result was a list of 44 promising requirement techniques, including for instance 9 variations of prototypes with usability testing.

Many well-known techniques were removed 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 [Conklin & Begeman, 1988]. They might be useful in other projects or during design, but our defect reports did not show a need for them.

The next step would be finding the best techniques. Ideally, we wanted to look for improved market value of the product, but we couldn't imagine reliable ways to assess it. The literature didn't give us much clue to the real-life benefit of each technique. Finally, we decided to look for net cost saving, i.e. saved work hours due to defect prevention or early detection, minus increased cost due to carrying out the technique.

3.2. Estimating hit-rates

As a first step, 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 or detect defect *d*. We were three experts looking at each defect to identify the techniques that might have revealed the problem, and estimate their hit-rates. We didn't take averages, but reached consensus for each hit-rate. To avoid hair splitting, however, we decided to use only hit-rates of 0, 5, 20, 50, 80, and 95%.

We realized that significant factors in the hit-rates were how well we expected developers to master the technique, how carefully the technique was applied, and how well the result was checked. As a side effect of these discussions, we improved the descriptions of the techniques.

For each defect, we typically found about five techniques that had some chance of preventing the defect.

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 defects that would have been found in the sample. Scaling up from the sample to the entire set of error reports gave the expected number of defects that would have been found in the entire project. As an example, we have these figures for the apparently strongest of our techniques, a specific variant of prototypes with usability tests:

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

Sum of hit-rates, h(230,d):	21.85 out of 200 defects
Expected defects found in total project:	87.40 out of 800 defects

In other words, in projects like NSL, we would expect technique 230 to find an average of 87 defects, 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 very different backgrounds and insisting on them reaching consensus rather than taking an average. A more detailed explanation of the hit-rates and the techniques considered may be found in Lauesen et al. [1996] and in Vinter et al. [1999].

3.3. Cost/benefit of 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, but with the rather short requirement specs we dealt with, it wasn't a critical issue.

Estimating the benefit of prevention turned out to be very hard, however. Many 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 one and two could be estimated in principle. Surprisingly, most defects were either considered unnecessary to repair or they cost just a few hours to repair. In many cases the repair was only partial or a work-around, because the kind of solution one would have made if realizing the problem early, was too costly at the late stage. An example is defect D1, as described in Section 2.

We realized that factors three and four 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 factor 1 and 2. As we will see in section 5.2, factor 3 and 4 were actually very significant.

The pragmatic decision was a prevention benefit for each defect of either 5, 25, or 50 hours. The total expected benefit if all requirement defects were prevented was as follows:

Total possible benefit:

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:	3,200 hours:	428 defects

3.4. Selecting Techniques

More than half of the techniques would be 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 find optimal combinations of techniques.

We spent much time computing optimal combinations, but observed that the filtering effect had little influence on the optimal combination. We had hoped that somehow a few techniques could combine like a jigsaw puzzle and detect most defects. This was not possible at all according to our calculations.

We realized that the optimal combinations were rather sensitive to small changes in benefit estimates. They also depended on the kind of project, for instance to what extent new external software was used, and to what extent the user interface was a large part of the project. Later we realized that the optimal combinations had little influence on what developers actually chose to do, because they had many other criteria that influenced their decision.

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.

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, but it wasn't visible in the defect reports, because they focused on defects in the product, not on lost market opportunities.

The end result was that we presented two development teams with a list of potentially good techniques. They decided to use five of them, and got training in them, but later realized that it was too many new things at once. 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 prototype screens (technique 220). This technique used prototypes with carefully designed screens and some screen switching features, but no real functionality. (Technique 230 used prototypes with some real functionality. It would prevent more defects, but would also discourage complete re-designs. As a result it was discarded).

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

Since the 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 the usability defects (13 out of 72 usability defects in our sample) and 11% of the others (functionality, interoperability, etc.).

Scenarios mean different things to different people [Campbell, 1992]. We do not use it in the object-oriented or UML sense, which is quite close to the computer solution, but as a more vivid description of the user environment and tasks (see the example below). Our approach is more like the ones reported in Carlshamre & Karlsson [1996], Graham [1998], and Hooper [1882]. For a description of usability testing in a developer setting, see Jørgensen [1990]. For a more full description of usability testing, see Dumas & Redish [1993].

4. 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. In spite of all, developers managed to get across the barrier.

Team A developed a portable sound intensity meter. 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 hardware and SW platform from another 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 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:

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) There was no easy way to carry the sound meter on the ladder. (2) The user had to answer a few dialog 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 feed-back. The display should be reversible programmatically, etc. 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.

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

Team B developed 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 product. Development then continued from there. The team overshot their budget significantly and didn't complete on time. However, we have not studied the project further.

5. Results

When team A had completed the project, we started analyzing 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.

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

1. **NSL:** The first product we studied. 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 first portable product. 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 portable product, developed by team A. The main focus in this product was on the usability side. The complexity of the screen pictures was comparable to the complexity of the windows in NSL.

Here are some figures for comparing the three projects:

	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**

5.1. 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 grabble 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-1 (due to many more 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.

5.2. Unexpected results

The most striking effect is that the new technique had reduced the number of usability problems per screen by about 70%. This is most likely the result of usability testing and willingness to redesign the interface. However, the effect is much more than the 18% reduction of usability defects expected in our expert predictions. It is even more surprising when you know 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 is at work.

A small effect is that the number of implementation defects per month is reduced by more than 20%. This was unexpected, but interviews with the developers gave a likely explanation:

The new approach created a solid foundation at an early stage. There was no doubt about requirements and user interface during the rest of 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.

As a result, PT-2 became the first and only one among scores of other proj??ects at B&K that was completed on time, without stress and with high customer satisfaction.

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 everywhere in the company.

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

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 21167). Without external funding, B&K had never commenced such a project.

The authors would like to thank Per-Michael Poulsen (the third expert), Kaj Ormstrup Jensen, and Jan Pries-Heje for their work and dedication while analyzing reports, studying the development process, and helping with ideas.

References

Beizer, B.: Software Testing Techniques, 2nd edition, Van Nostrand Reinhold, 1990.

Campbell, R.L.: Will the Real Scenario Please Stand up? SIGCHI Bulletin, April 1992, pp. 6-8.

Carlshamre, P. & Karlsson, J.: A Usability-Oriented Approach to Requirements Engineering. Proceedings of ICRE'96, pp. 145-152. IEEE Computer Society Press, 1996.

Conklin, J. & Begeman, M.: gIBIS: A Hypertext Tool for Exploratory Policy Discussions. ACM Trans. Office Inf. Systems 6(4): pp. 303-331, 1988.

Dumas, J.S. & Redish, J.C.: A practical guide to usability testing. Ablex 1993.

Graham, I.: Requirements Engineering and Rapid Development. Addison Wesley, 1998.

Hooper, J.W. & Hsia, P.: Scenario-Based Prototyping for Requirements Identification. ACM SIGSOFT, Software Engineering Notes, vol.7, no 5, Dec. 1982, pp. 88-93.

Jones, C.: Applied software measurement. McGraw-Hill, 1991.

Jørgensen, A.H. (1990): Thinking-aloud in user interface design: a method promoting cognitive ergonomics. Ergonomics, 1990, Vol. 33, No.4, pp. 501-507.

Lauesen, S., Poulsen, P.-M. & Vinter, O. (1996): Experience-based requirements engineering. In: Graeme Shanks & Paul A. Swatman (eds.): Proceedings of 1st Australian Requirements Engineering Workshop, Monash University, Melbourne, 1996, ISBN 0 85590 106 3, pp. 13.1-13.9.

Vinter, O., Lauesen, S. & Pries-Heje, J. (1999): 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. WWW: http://www.esi.es/VASIE/Reports/All/21167.