JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2005; **17**:277–306
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.313

Practice

Design preservation over subsequent releases of a software product: a case study of Baan ERP

Jilles van Gurp[1,*,†], Sjaak Brinkkemper[2] and Jan Bosch[3]

[1]*Creative Online Development B.V., Nijmegen, The Netherlands*
[2]*Institute of Information and Computing Sciences, University of Utrecht, Utrecht, The Netherlands*
[3]*Nokia Research Center, FI-00045 Nokia Group, Finland*

SUMMARY

We present the results of two case studies we conducted at Baan in the Netherlands. At the time of conducting the case studies, Baan was part of Invensys plc. (Baan is now owned by SSA Global Technologies.) In these case studies we investigated how companies identify design erosion and address this in their software, a practice we call 'design preservation'. In this study, we selected two sub-systems in Baan products that had recently been subjected to extensive maintenance activities because they were eroded. In this paper, we analyze the problems these systems had, how Baan identified that these systems were problematic, and the remedies that were used to address the problems. In addition to confirming some of our earlier conclusions, we have been able to extract some common causes for design erosion problems as well as a number of recommended design preservation practices, which, at least for Baan, have proven to be very effective in strengthening design preservation. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: software aging; design erosion; architecture erosion; software quality; software evolution; enhancive maintenance; adaptive maintenance

1. INTRODUCTION

1.1. Software erosion and design preservation

The growing scale of software systems makes it increasingly hard to maintain software due to the amount and complexity of the source code. However, at the same time, it is also becoming ever more infeasible to discard large software systems due to the investment these systems represent.

*Correspondence to: Jilles van Gurp, Creative Online Development B.V., Wijchenseweg 111, NL-6538 SW Nijmegen, The Netherlands.
†E-mail: jilles@jillesvangurp.com

Received 6 March 2004
Revised 15 February 2005
Accepted 17 February 2005

However, software-developing organizations often find themselves in a situation where their existing software or sub-systems of their software are proving to be increasingly hard to maintain and adapt to new requirements. A major concern for software developing companies is that this situation forces them to choose between two evils: either abandon a software system representing years or decades of work and invest heavily in a new system to replace it, or continue to maintain a non-maintainable system.

Neither option is very attractive. The first option would essentially set such companies back a few years (time they need to re-develop their software). Potentially (e.g., if competing companies have better software) this could threaten the existence of such companies. The other alternative (continue to maintain) is not attractive either, since elevated maintenance costs or an inability to adapt the software to new customer requirements deteriorate the competitive advantage that the software provides. Ultimately, this leads to a situation where other players on the market have a much stronger software offering. Because the only options are either to completely replace the system, or to subject the system to an extensive revision, there is not much that can be done about such a situation on short notice.

Web browser developer Netscape is an example of a company that got itself in a situation where they were faced with this choice [1]. In 1997, version 4.0 of Netscape Communicator was released. Around the same time, Microsoft released its version 4.0 of Internet Explorer. The subsequent events are well known: the two companies engaged in a browser war, which ended with Microsoft dominating the browser market. Around 1998 it was becoming apparent that, for various reasons, Netscape Communicator was losing the battle. Netscape urgently needed a new and improved version of their Netscape Communicator product. However, they could not deliver it. The old version 4 source code was proving hard to adapt and finally in late 1998, Netscape, in a desperate effort to regain its market share, chose to release its browser as open source (in what is now known as the Mozilla project). After a few months, the old source code was discarded and development of a new browser was started. Mozilla 1.0 was released June 2002. During the development of Mozilla, Netscape continued to release minor updates to Netscape Communicator and even released an ill-fated version 6.0 based on a development release of the Mozilla project. However, their market share shrank from more than 50% to what is now estimated at less than 5%. While recent reviews of releases of the Mozilla browser (and commercial Netscape releases derived from it) are quite favorable, Microsoft continues to dominate the browser market. Recently, what remained of the company Netscape was liquidated. The Mozilla project is now managed by an independent foundation.

We believe that problems such as encountered by Netscape are common to all large software product vendors. As their products evolve in subsequent releases, changes accumulate and may become an obstacle for further releases. This phenomenon has been referred to as architecture erosion or software aging [2–4]. We prefer the term design erosion to architecture erosion because, in our earlier case study [5], we found that the effects of design erosion are not limited to just the software architecture, but affect all development and maintenance artifacts. The case study we present in this paper confirms this view.

Design erosion is the cumulative, negative effect of changes on the quality of a software system. By quality we refer to the aggregated quality attributes such as extensibility, flexibility, maintainability, etc. Each change is a trade-off between cost, requirements (functional and non-functional) and time (e.g., product release deadlines). Inevitably, compromises have to be made with respect to some quality attributes. As a result, the quality of the system is said to erode because of the combined and cumulative effects of such compromises over time.

This case study was further motivated by our experience with various other case studies we have conducted with other companies in the past decade. For example, Axis AB in Sweden [6] who manufacture various server appliances (e.g., printers or scanners) to connect to a network, have a reusable software system that they use in all their products. At some point, they found out that certain features that were required could not be added to their existing software. The only way to resolve this situation was to redevelop the software. This took more than two years, during which they continued to release products based on the old software and simultaneously worked on the next generation of their software.

Design erosion is a wide-spread problem that, as we argued in our earlier study [5], eventually affects all software systems. In [5], we presented our experiences with the evolution of a software system that we created. Our software system was evolved in a number of steps. In each evolutionary step, new requirements were introduced and the software system was changed to meet these requirements. The study clearly demonstrates that new requirements may conflict with design decisions taken earlier. These conflicts either lead to radical changes in the software or, if that is infeasible, result in awkward design solutions that work around rather than fix the problems. In the latter case problems with respect to understandability, flexibility, and maintainability may arise since the changed design is not optimal for the set of requirements it must address. Due to subsequent sub-optimal design decisions, these problems only get worse.

Obviously, design erosion poses an enormous threat to any organization that depends on large software systems. These organizations, at the very least, risk losing their competitive advantage because they gradually lose the ability to make necessary changes to their systems. Ultimately, the existence of such organizations may be threatened as well since it may prove very expensive to repair eroded software systems. The Netscape case is a good example of this. Note that this applies both to software product vendors that commercially depend on the quality of their products, and to organizations owning large scale software systems supporting critical business processes.

Consequently, it is important for companies to be able to recognize signs of design erosion before it is too late (i.e., before facing an expensive repair/replacement choice). If design erosion is detected in time, an effort can be made to preserve the software design. Design preservation is about ensuring that an evolving software design remains in such a condition that necessary changes remain possible. Typically, as is illustrated by the case studies presented in this paper, *design preservation* is an ongoing activity to recognize, repair, and prevent design erosion that includes five aspects.

- *Symptoms*. Monitoring the quality attributes of the software during its life.
- *Identification*. Identifying low-quality software components.
- *Causes*. Analyzing why the quality is low.
- *Resolution*. Addressing the quality issues by executing corrective actions.
- *Prevention*. Adapting the development and maintenance processes in order to avoid these problems in the future.

1.2. Research questions

In this paper, we present the results of two case studies, which we conducted at Baan to investigate how such a large software developing organization has managed to preserve the design of its software product. Baan, currently a part of SSA Global Technologies, is an international software company that develops ERP (Enterprise Resource Planning) products and implements these products on site.

Like many organization fielding software products, Baan regards most software maintenance work, such as enhancive, adaptive, or perfective maintenance, as being continued development, and hence usually terms most maintenance as 'development'.

Baan's ERP software consists of a set of integrated packages for the administrative support of manufacturing, resource planning, sales, purchasing, warehousing, and finance. About 15 000 customer sites worldwide are running Baan ERP software in the larger and medium enterprises. In the ERP domain, Baan has been competing with companies such as SAP, Oracle, Peoplesoft, and Microsoft Navision. Baan has a large research and development department with development centers performing development and some types of maintenance in the Netherlands and India [7]. The research and development department is responsible for the Business Intelligence, Middleware, and E-commerce products in the ERP product line, which are the subjects of our case studies.

The purpose of these case studies is to explore the problems and issues encountered in software developing organizations, such as Baan, with respect to design erosion and to design preservation strategies. In order to do so, we answer the following research questions (derived from the five aspects of design preservation outlined above).

- *Quality monitoring*. What are the effects of design erosion on a system?
- *Identification*. How does an organization decide that their software is eroded and needs to be repaired and how does this decision process work?
- *Analysis*. What are common causes for erosion?
- *Resolution*. What kinds of solutions are applied to fix an eroded system? How and when do decisions with respect to preservation and repair need to be taken?
- *Prevention*. What practices help prevent erosion? What are good practices that are applied in both cases?

In Section 5, some generalized answers to these questions based on both case studies are presented. Additionally, based on the current design practice at Baan, a number of recommended practices are proffered.

1.3. Organization of this paper

In Section 2, we describe our research method. The case studies are presented in Sections 3 and 4, respectively. In Section 5, we provide answers to the research questions and discuss them in relation to related work. We conclude our paper in Section 6 with the lessons learned.

2. RESEARCH METHOD

In this section, we outline the empirical research approach we have applied in the case studies and discuss its strengths and weaknesses. In his editorial for a journal of empirical software engineering [8], Basili makes a plea for the use of empirical studies to validate theories and models that are the result of software engineering research. In a more recent publication, [9], Basili presents an overview of how empirical research has benefited NASA's Software Engineering Lab. When performing empirical research, a distinction can be made between qualitative empirical studies and quantitative studies.

The approach advocated by Basili in [8,9] can be characterized as mostly quantitative. As can be seen in [9], collecting quantitative data is a labor-intensive process that needs to be tightly integrated with the software processes used. In a setting such as NASA, where reliable, dependable software is required, this is feasible. The results of the quantitative empirical research are used to optimize the maintenance and development processes. However, in many other contexts this is much less feasible.

Qualitative data, on the other hand, is relatively easy to obtain and has the advantage of providing more explanatory information [10], which, in an exploratory case study such as ours, is very desirable. As is noted by Seaman in [10], neither quantitative nor qualitative empirical research can prove a given hypothesis. Empirical research can only be used to support or refute a given hypothesis. According to Seaman, a combination of both quantitative and qualitative studies is the best way of supporting a hypothesis. In this paper, we try to combine the best of both worlds.

2.1. Case selection

Throughout both case studies, we have cooperated with Baan's Research and Development (R&D) department. Baan R&D management was very much interested in the results of the case study for the sake of: (a) providing an outsider analysis on the architecting and engineering practices; and (b) educating the product architects and software engineers with the results. Our primary contact there was the process architect (at the time of the case study), Sjaak Brinkkemper, who is also a co-author of this paper. Using the expertise and knowledge of Baan's product portfolio, two representative sub-systems were selected for further study, and contacts with the staff working on these sub-systems were initiated. Before selecting the sub-systems for the cases, we had several meetings with the R&D department during which we discussed the organizational structure, the product lines of Baan, the goals for the case study, and made an estimate of the time needed for both case studies.

We were looking for software products with the following properties:

- the systems had to be mature enough to have endured design evolution;
- during the evolution, there must have been significant changes in the requirements;
- it should be possible to both interview people who were involved in the initial development of the system and people who were involved in restructuring the system for new requirements.

2.2. Interviews

In this exploratory case study, we use interviews as the primary tool of retrieving information. Consequently, our research is mostly of a qualitative nature. However, where possible, we complement the qualitative data with quantitative data provided by the interviewees (e.g., estimated defect rates, number of lines of code, etc.). Much of the qualitative information we extracted from the interviewees is actually based on quantitative information available internally within Baan. However, due to the confidential nature of such data we were not given direct access. We have experienced that, in general, software-developing companies are reluctant to disclose quantitative data on defect rates or any other metrics that might be used to adversely affect competitive stance.

In both case studies reported in this paper, the interviews followed the same pattern. We first met with the interviewees in a group for an introductory meeting. During this meeting, the purpose of the case study was communicated and a brainstorm session was held to select appropriate modules/components

for further study. This meeting was also used for planning subsequent interviews. In the following meetings, both group and individual interviews were held during which more specific questions about the design and evolution of the system were asked. In addition to interviews, we were given access to various documents including, for example, functional designs and requirements documentation. Using these documents, we were able to both verify/clarify certain statements of the interviewees as well as prepare specific questions in advance.

2.3. Validation

To ensure the correctness of our data and conclusions, we have used two methods.

- *Cross checking*. In both cases, we interviewed multiple personnel separately. This allowed us to compare their answers and verify whether there were any contradictions. In both cases, we were given access to relevant software documentation, which allowed us to further validate our information.
- *Feedback*. An important part of qualitative research is feedback. The data presented in this article consists mostly of our interpretation of interviews with the interviewees. Therefore, verifying whether or not this interpretation is correct is an essential part of ensuring the validity of our case study. After each meeting, a report detailing our conclusions and interpretation was communicated to the interviewees for feedback. In addition, this paper was co-authored and the case study results were reviewed by a member of Baan's R&D department and several other people within Baan.

2.4. Limitations of this study

There are a number of issues with our research approach that may affect the validity and generality of our findings.

- *Representativeness of the cases*. By limiting ourselves to one company, we risk that these case studies' conclusions may not be applicable to other domains and companies. Both the corporate culture and the domain Baan is operating in affect our conclusions. However, based on our experience with case studies in other companies, the corporate culture in Baan is representative of many software-developing companies. Additionally, Baan is one of the market leaders in their domain and can thus be seen as representative for its domain.
- *Quantitative data*. As explained earlier, we use a (mostly) qualitative approach. Complementing our data with quantitative metrics would strengthen our conclusions. However, there are a few reasons why this study does so only to a limited extent. First of all, this is an exploratory study. A quantitative study requires a precise formulation of hypotheses, relevant quantifiable parameters, and a model for the interpretation of values for these parameters. A study such as presented here may provide the necessary input to formulate hypotheses and parameters for future quantitative studies. Second, a quantitative study generally requires a significant investment of time and other resources. Finally, many relevant metrics that would need to be collected are generally considered as sensitive information in software development organizations. Baan is no different in this respect.

- *Cases are not comparable*. We have deliberately chosen to research two cases from different domains to show that identification, resolution, and prevention of design erosion works the same across domains. Therefore, however, both cases use different types of technology and involve people with different skills and training. On the other hand, both teams operate in centrally managed release projects to design and build the sub-systems as part of one product as that software evolves. This makes it possible to compare the results of both case studies, notwithstanding some limitations.
- *Biased response from interviewees*. Because of their involvement with the software, a potential problem may be that the responses from the interviews are biased. However, we do note that many of the interviewees had not been involved with the earlier versions of the software about which we were interviewing them. A potential other concern is the fact that our interviewees might be biased by the fact that the co-author held a management position within the R&D department. In practice, however, this proved not to be an issue. Baan R&D is so large that the department our co-author was in has little to do with the departments where we conducted our case studies. In fact, when meeting with the interviewees of the query processor (QP) case, our co-author actually had to introduce himself to the interviewees since that was the first time they had met.

3. CASE 1: THE QUERY PROCESSOR

The QP is an important sub-system component in the database access layer in the Baan ERP system product line. The database layer has evolved for more than a decade and currently supports several commercial Relational Database Management Systems (RDBMSs) based on SQL (Structured Query Language). However, when it was originally created, SQL was not even standardized. In addition, the way applications interacted with a database was very different from how this is done today. Consequently, the database layer has seen many changes during its existence.

Early versions of the database layer did not support SQL at all (and did not have the QP component). Applications interacted with the database through a procedural application programmer interface (API) known as C-ISAM (C Indexed Sequential Access Method). This procedural API allows for operations on rows of database tables. Over the years, the database layer was changed to support SQL. The QP component was originally introduced as a means to convert SQL to C-ISAM calls and vice versa. However, over the years this component has been changed considerably to support new SQL features, SQL RDBMSs, etc. In 1999, it was decided to redesign and re-implement the QP component. In this section, we examine how the QP component evolved, why it needed to be replaced, and how the new version addresses the issues with the old version.

3.1. Evolution

The set of modules that offers, among others, the database functionality in the Baan ERP product line is known as Tools. Before version 4.3 of this package, the Baan software ran on a central mainframe computer with terminals attached. Baan applications executed in the Baan shell which, in turn, simply used the procedural C-ISAM layer to communicate with the database (see Figure 1). Note, that the special shape of the Baan applications in the figures relates to the modular ERP packages for Sales,

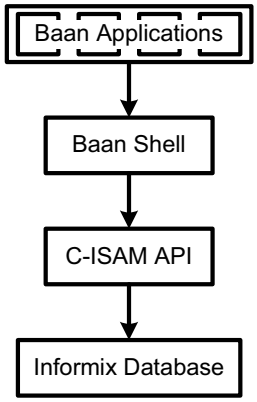


Figure 1. Overview of Baan architecture before version 4.3 of the Tools package.

Manufacturing, Purchase, Warehousing, and the like. However, we do not go into more detail as this generic functional architecture is beyond the scope of this paper.

Around 1990, the architecture was converted to a client–server architecture. Rather than calling the database directly through the C-ISAM API, operations on the database were passed to a component known as BDB (Baan Database), which has a client and a server part. The BDB component has a similar API as C-ISAM so for applications the changes were relatively minor. The BDB client (BDB/C) component passes the calls to the BDB server (BDB/S) component (see Figure 2). BDB/S in turn passes the calls to the DB-Driver component, which in turn calls a database specific driver (e.g., Oracle or Informix). The purpose of the DB-Driver component is to shield BDB/S from any database vendor specific issues. Consequently, Baan applications can work with databases from multiple vendors. In addition to passing database calls, the BDB components also provide functionality for transactions and sessions, i.e., the so-called ACID (Atomicity, Consistency, Isolation, and Durability) properties [11].

Shortly after the move to the new architecture, there was an increasing demand from application developers to support SQL. SQL was standardized in 1992. However, at this time there were still significant differences in what parts of the standard were supported by database vendors. In addition, there were differences in how specific SQL features were supported, and many databases have proprietary extensions to SQL.

To prevent creating database-vendor specific applications, Baan SQL was created, which supports a subset of SQL features deemed important as well as some Baan specific features and extensions of SQL. A new client-side module, the QP, allowed applications to use Baan SQL to query the database. In the first version, QP simply translated SQL to database calls and subsequently called the BDB/C component to execute the calls. Since this is done on the client side, we will refer to this component as QP/C (see Figure 3).

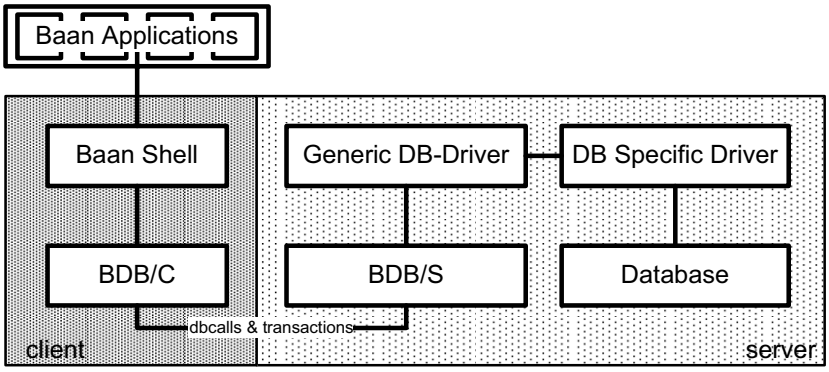


Figure 2. Baan architecture from and after version 4.3 of the Tools package.

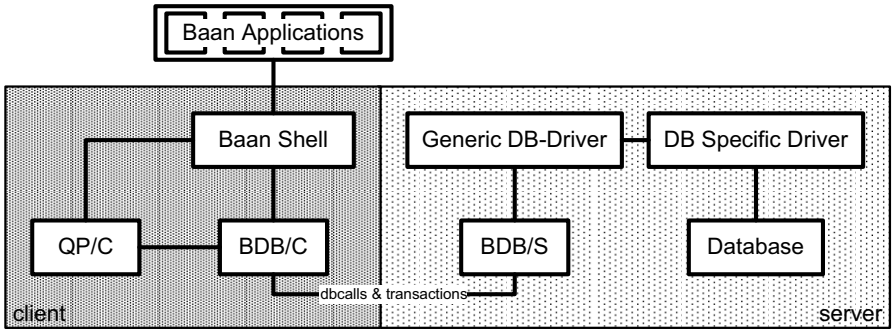


Figure 3. QP/C added SQL on the client side.

In the next version (Figure 4), the server side was also equipped with a QP module. This made it possible to reduce traffic between client and server. Because of the row-based nature of database calls, there was a lot of network traffic between client and server. By moving part of the evaluation of a SQL query to the server, the amount of traffic could be reduced. A client-side QP component was still needed, e.g., for doing joins on query results from multiple databases. In addition, QP/C contains functionality to work around certain database specific limitations (such as, e.g., a maximum amount of allowed joins or a maximum amount of columns in a table).

In this version, QP/S still used BDB/S for communication with the database. The database drivers in this generation of the architecture are referred to as level 1 drivers to indicate that they are still row oriented (i.e., SQL statements are translated to C-ISAM style database calls).

The intention of a large customer to deploy Baan applications in a Wide-Area Network (WAN) prompted Baan to eliminate database calls between client and server entirely because the latency

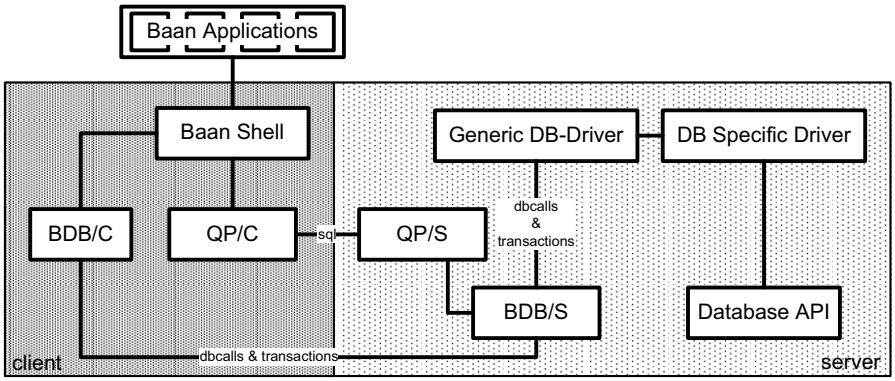


Figure 4. QP/S added SQL on the server side.

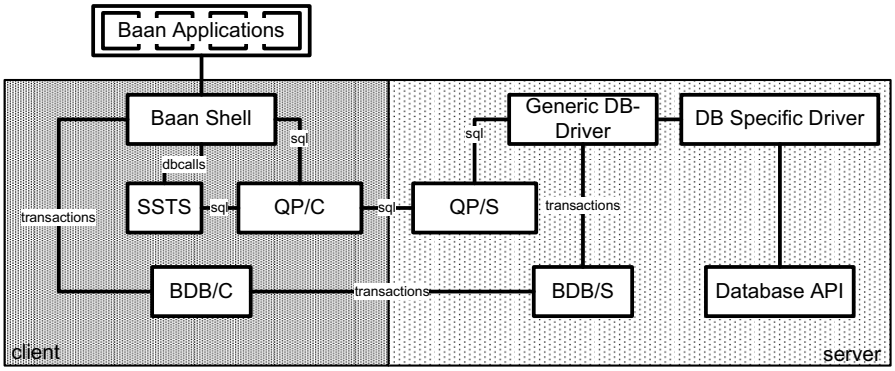


Figure 5. Baan architecture with the addition of the SSTS and the level 2 driver.

introduced by using a WAN would make these calls too expensive. Instead of database calls, all database interaction was now done using SQL (see Figure 5).

However, database calls still needed to be supported because of the large number of applications using this style of interacting with the database. Consequently, database calls needed to be translated to SQL on the client side. This was done by adding the SSTS (Single row, Single table Translation Services) component to the QP/C as shown in Figure 5. The BDB/C module was still needed though to provide functionality related to sessions and transactions.

Up until this version of the architecture, all queries had passed through BDB/S, which communicated with a database driver (referred to as a level 1 driver) component through the procedural C-ISAM API. However, modern databases support all or most of SQL 92 and many newer features that were added to

SQL later. Converting this advanced type of SQL to the procedural calls used by the driver component is not feasible. Therefore, a second type of database driver was added to the architecture that can convert the Baan version of SQL to a database specific SQL without the conversion to the C-ISAM calls. This new type of driver was referred to as the level 2 driver.

Additionally, the level 2 drivers still had to support Baan SQL specific things (e.g., array columns, column lengths greater than supported by the underlying database, a Baan-specific LIKE operator, etc.) that were not supported in regular SQL. With the level 1 drivers, this was relatively easy; however the level 2 drivers needed to translate to SQL instead of procedural calls. The QP/S became responsible for splitting Baan SQL queries into regular SQL sub-queries that can be understood by the level 2 driver and then later combine the results of these sub-queries.

Despite the fact that no C-ISAM style calls are needed anymore, the BDB client and server components are still needed for handling the transaction logic (i.e., making groups of queries atomic). This is especially important if more than one database is involved in a query (e.g., a join of tables from different databases).

In incremental steps, the architecture evolved from a single tier architecture with support for C-ISAM databases (Figure 1) to a client–server architecture with support for modern SQL databases (Figure 5). During the process, the system became increasingly difficult to maintain (more on this later) and thus it was decided to redesign and develop both QP components.

3.2. Problems and causes

There were several issues with the QP/C and QP/S components that led to the decision to redevelop these components. A distinction should be made between the symptoms and their causes. Important symptoms that something was wrong with the QP components include the following.

- It was observed (during routine inspections of defect metrics, etc.) that fixing defects in the QP components took unusually long. The personnel we interviewed estimated that the average time that was needed to fix a typical defect was about a week.
- The list of defects that needed to be fixed had grown quite substantially. About 100 serious defects had been identified. Fixing some of these defects would require a substantial redesign of the QP components. Addressing all defects would require substantial investment.
- The process of fixing a defect often introduced new defects and regressions.
- The behavior of the QP components was inconsistent with the SQL standard or any other form of documentation available internally.
- Affected personnel had learned to work around these inconsistencies and were actually relying on these workarounds for the correct behavior of their applications.
- Personnel found it difficult to understand the design of the QP modules. After the personnel who developed the initial version of QP/C left the company, this became a problem.
- Poor code quality. Due to the various changes, the lack of documentation and the lack of design, the code quality had deteriorated over the years. The personnel cited a lack of modularity, dependencies between code modules, and pollution of data structures with unrelated data elements, as being the most striking symptoms of this problem.

A number of causes for these issues were identified as follows.

- Over the years, there had been a number of changes (as discussed above) due to considerable evolution in requirements or due to evolution of database technology. From a procedural C-ISAM-based, single-tier approach, the system evolved to a client–server architecture with support for SQL on the client. Subsequent changes also introduced a RDBMS with SQL on the server side and finally the C-ISAM style of database interaction was replaced by SQL entirely. Both QP modules were substantially affected by these changes.
- The behavior of the QP evolved beyond what had been envisioned when the QP had been designed. Consequently, up-to-date architecture design documentation and other specifications were lacking.
- The lack of proper documentation made it hard to test the QP components since specifications of the correct behavior were lacking. Consequently, users of the component had to find out for themselves whether and how the component worked, which explains why application maintainers were relying on incorrect understandings of the behavior of the component.
- The people who originally designed the QP were no longer working for Baan. The system they created was very complex and hard to understand for the people who replaced them.
- QP/S supported both the obsolete level 1 and the new level 2 drivers. The functionality for these two types of drivers was mixed. Many problems originated from the operational differences between level 1 QP and the level 2 QP.

3.3. Solutions

As pointed out previously, the QP components were below company standards in 1999. They had become increasingly hard to maintain. There were numerous defects that were scheduled to be fixed and the personnel were increasingly depending on incorrect understandings of the behavior of these components and working around the components' limitations. In addition, SQL had by then become the standard way of interacting with databases and there was an increasing demand from users and potential customers for supporting the more advanced features of SQL such as, for example, unions, expressions in `select` clauses, the distinct key word, etc.

To address these issues, a project was started to redesign the QP components. An explicit goal was to make the transition to the NQP (new QP) as smooth as possible and to prevent having to perform adaptive maintenance on the existing applications [12]. However, at the same time it was recognized that incorrect usage of SQL (which was permitted with the old QP) could no longer be tolerated. Consequently, some degree of incompatibility was anticipated.

As a result, the existing architecture was preserved and only the QP components were replaced. It was anticipated that it was not possible to complete all of the NQP before the next Baan product release. So, it was decided to break up the work in two stages. In stage 0, the QP/S was replaced, and in stage 1 the QP/C was replaced as well. Both releases had a full functioning query processing, although the underlying architecture changed significantly. In addition, the careful planning for the migration resulted in a minimum amount of effort required for adapting the Baan applications portfolio. The switch, which had been built into the NQP to optionally use the old QP, could be removed once stage 1 was completed.

A number of practices were adopted to prevent the NQP from suffering the same fate as the old QP.

- *Adoption of object-oriented principles*. It was recognized that certain object-oriented features such as, for example, inheritance, delegation and information hiding, were needed to improve reusability and modularity of the system. Therefore, it was decided to use C++ for the NQP rather than ordinary C, which was used in the rest of the system. However, for portability reasons (in 1999 there were still considerable differences between C++ compilers on various platforms) only a subset of C++ features was to be used (for example, C++ templates are not used).
- *Explicit requirements and design*. Since the early 1990s, a number of development and maintenance practices had been adopted in Baan. Among them were rigorous requirements and design phases [13]. Therefore, before starting the realization phase, the NQP requirements were specified and an object-oriented design was created. These documents also formed the basis for end-user documentation and regression tests.
- *Documentation*. The SQL supported by the NQP was fully documented so that there could be no misunderstanding as to how to use it. Any inconsistency between documentation and actual behavior is now considered to be a defect.
- *Automated regression testing*. For each encountered defect and for each feature, an automated test was created to prevent future regressions by subsequent changes. By the time stage 0 was complete, the test suite consisted of about 330 automated tests, which were run after each change to the system. After stage 1, the number had grown to about 800 tests.
- *Migration tools*. Because incorrect QP behavior was no longer supported, applications needed to be migrated to the NQP. During a transition period, personnel could indicate with a flag whether the old QP or the NQP was to be used. In addition, they were given tools that automatically located SQL constructions in their applications that were no longer supported. After stage 1 was finished, the ability to run the old QP was disabled (i.e., the recent versions of Baan applications use the NQP).

3.4. Analysis

At the time we conducted our interviews, a version of the Baan ERP product line had been completed that included the NQP (stage 0 and stage 1). The project was considered successful by the personnel involved and the management. The following aspects were cited as success indicators.

- The defect fixing time had decreased considerably. In the old QP a single defect often took more than a week to fix. In the new QP, defects were fixed much faster (typically less than a day).
- There were considerably fewer defects. Interviewees even claimed that there had so far been hardly any need for corrective maintenance. Most likely, this is because of the regression tests which automatically test most of the software's features. By the time NQP/C was finished, there were 800+ automated tests that were run after every major change, which helped prevent the incorporation of additional defects from maintenance.
- The migration was very successful. Migrating the old Baan applications to the NQP had been an important priority. The migration was so successful that the objective was reached to remove the old QP after stage 1 was completed.
- Due to the increased modularity of the software, it is easier to implement new features. Interviewees indicated that identifying and implementing new SQL features was their current priority.

Also during the implementation of stage 1 they were able to reuse classes from the stage 0 implementation.

Arguably, the NQP is significantly better than the old QP component in terms of extensibility and code quality. However, that does not mean it is free from issues. During implementation of NQP stage 1, personnel encountered a problem with some of the code in the stage 0 implementation. Under time-pressure, this code was not as generic as it should have been to allow for reuse in stage 1. Consequently, some adaptive maintenance was needed to fix this. In addition, there were some integration problems of NQP/C with the Baan shell. In the end it was decided to support some of the legacy features of the old QP to prevent breaking compatibility with existing applications. However, these legacy features are now fully documented (so there are no misunderstandings about the NQP behavior) and their use in new applications is discouraged.

4. CASE 2: PURCHASE AND SALES SCHEDULES

The subject of our second case study is the Purchase and Sales Schedules (PSS) functionality in the Order Management (OM) package, which is one of the application subsystems in the Baan ERP system product line. Due to the different nature of this application and the different type of information we retrieved from the interviewees, we use a slightly different format than in the previous section. In the previous section, we looked at multiple versions of the QP component to reconstruct its evolution.

The second case concerns evolution within only one version of the system. Our intention is to demonstrate that design erosion can also occur within the time-span of one version. In addition, it should be mentioned that the maintenance of this version spanned a considerable amount of time during which various test versions were tried out at customer sites. So, even though this is a single release, the software and artifacts underwent a significant amount of changes, some of which were specifically aimed at improving the quality of the system.

Because we are only looking at one release, we do not present an overview of the architecture evolution as in the previous section. Instead, we present a brief overview of the important milestones in the development of the PSS package.

Like all applications in the Baan ERP product line, the Order Management system is an application that runs on top of an application engine, called the Baan shell. The Baan shell is a virtual machine that provides maintainers with a C-like 4GL language and sits as a layer between Baan applications and the infrastructure (e.g., the database functionality discussed in the previous section). The Baan ERP architecture is outlined in Figure 6. In this figure, the gray boxes represent the components that are developed by third parties. As mentioned, the Baan shell environment works as a virtual machine to ensure platform independence. The architecture is portable and is able to run with various operating systems/user interfaces. The application code consists of a number of packages, which in turn consist of modules (these are also referred to as dlls within Baan). The PSS case study mainly deals with one of these packages: the OM package.

PSS are a highly automated way of ordering without a lot of administrative handling based on long-term supplier agreements, which sits aside of normal ordering processes within OM (see Figure 7). A schedule is a list of requirements containing both forecast and order information that is send to the supplier on regular intervals. The forecast contains future global order information.

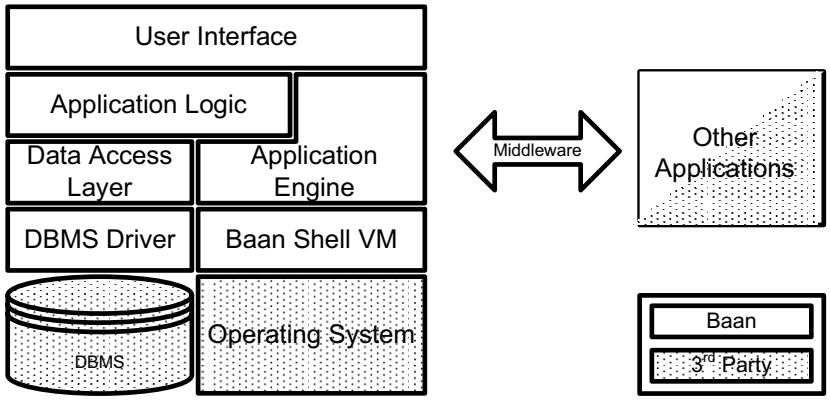


Figure 6. General overview of the Baan ERP architecture.

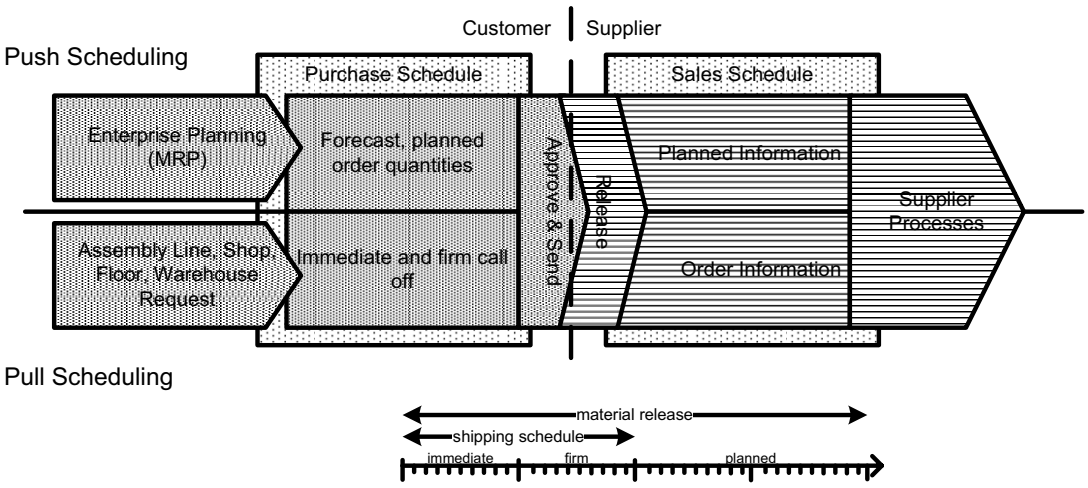


Figure 7. Push and Pull scheduling.

The order information consists of detailed data on delivery date, quantity, and sequence. This data is retrieved from a planning engine (Push Scheduling) or from decentralized demand (Pull Scheduling). The supplier can use the forecast information for its production planning and use the order information for delivery planning. The goods are delivered to a central warehouse (Push-based) or directly to the place where the demand was raised (Pull-based).

These types of schedules are used in industries where the goods flow is intensive, a large degree of tuning is required, and timely execution is crucial for further operations. This is called the Just-In-Time (JIT) paradigm in manufacturing. Boeing, which is an important customer of Baan, is a good example of this. The parts that are used to construct the airplanes arrive just in time on the construction site. For one and a half years, Boeing has been using the Baan PSS package to coordinate its JIT processes.

4.1. Evolution

In Baan IV, Sales Order Delivery Schedules were supported. However, the need was identified to address JIT principles and Original Equipment Manufacturer (OEM) relationships. To address these requirements, the PSS package was added to the OM application. An internal project for the realization of the PSS functionality was started as part of the development of the iBaan ERP 5.0c version.

First, the requirements were analyzed with, amongst others, DAF Trucks (a Dutch truck manufacturer). After this, a conceptual solution was designed and a prototype was developed by one of the development departments of Baan. The design and prototype were reviewed and tested by DAF Trucks. In the remainder of this paper, we will refer to the prototype as PSS1.

Because of the feedback returned by DAF Trucks and other customers, it was not possible to complete the realization phase in time for the release of iBaan ERP 5.0c. The feedback included a substantial amount of new requirements that proved hard to implement in the prototype. The PSS1 design was not capable of absorbing these additional requirements, which resulted in a number of quality problems (see the next section). Hence, there was a need to make drastic design changes and extend the project for another year. The main focal points in this project extension were integration with other ERP packages and code quality.

Even though the prototype was part of the 5.0c release (mainly to allow specific customers to evaluate it), it was not made available to the Baan 5 customer base because of the listed problems. However, Baan used the feedback on the design and prototype to further develop the PSS concept in iBaan ERP 5.1a. This release (PSS2) was a beta for the later iBaan ERP 6 series, focused on a single customer: Boeing. Boeing added more requirements, reviewed the designs, and tested the intermediate solutions. Subsequently they went on to deploy the iBaan ERP 5.1a on multiple sites in a live environment. This proved to Baan that this version was mature enough to be shipped with the generally available iBaan ERP 6 release. At the time that this paper was prepared, the PSS2 has been deployed at Boeing for nearly one and a half years without any significant problems.

4.2. Problems and causes

As discussed above, there were a number of problems with the PSS1 module that prevented its deployment in the iBaan ERP 5.0c release. Before starting development on PSS2, Baan conducted an internal evaluation to assess what needed to be done. This evaluation revealed a number of problems. The PSS package had been developed under a lot of time-pressure, and subsequent requirement changes had resulted in a poorly structured system that affected the quality of the source code. Specific problems that were cited by personnel included the following.

- *Code quality problems*. Most of the application code was programmed in a C-like 4GL language. While very flexible, this language is also easily abused in use. A number of problems with the code quality were cited in relation to PSS1. These included the use of global variables, confusion with respect to input and output variables (output variables were sometimes mistaken for local input variables), and initialization problems. A consequence of these problems was that the code was unnecessarily complex. Additionally, the number of defects was very high.
- *Database Access Layer*. The application architecture in Baan includes the Data Access Layer (DAL) that shields applications from the database (see Figure 6). This layer contains business objects and functionality related to the persistence of these objects in so-called DAL scripts. The addition of this layer to the architecture is relatively new and its use was not enforced during the PSS development. Consequently, there was a number of problems with application code that accessed the database directly. Also there was a number of cases where DAL scripts were extended with user interface code.
- *Modularity*. As mentioned before, Baan applications are decomposed into packages and modules. A problem with the PSS modules was that there were many dependencies between modules. In addition, several modules were very large (more than 10 thousand lines of code (10 KLOC) per module in some cases) and functionality for some features was spread over multiple modules. In addition to making the system more complex, this also caused problems with the source code management system. When someone checks out a module from the source code management system, it is locked (i.e., other personnel cannot make modifications to it). Because there were many dependencies and because some modules were very large, personnel typically had to check out a large number of modules to work on a particular feature. Consequently, personnel had to wait for each other because files they needed to modify were checked out to others making different modifications.
- *Defects*. Both project data about the number of revisions and the test data indicated that there were an unusually large number of change requests, defects, and revisions in the PSS1 prototype. An additional problem was that fixes for these defects were not always documented properly. Consequently, the design documentation did not reflect all the changes to the system.
- *Performance*. There were various performance problems with the data model. In a database centric product like PSS, the data model is very important. The data model was created as part of the design phase. One of the problems interviewees cited was that the model was not normalized. The normalization of database tables is necessary to restrict storing redundant data in the database [11].
- *Documentation*. The documentation was incomplete (it was mostly based on the initial design of the PSS1), contained errors, and generally lacked detail. Consequently, in many cases, the correct behavior of the system was not specified and the actual behavior not described.

Some common causes for these issues are as follows.

- *Requirement changes*. The PSS1 design was based on the initial requirements that were gathered from various sources. However, during the development more requirements were added by among others, DAF Trucks. This resulted in a large number of change requests.
- *Communication issues*. The PSS1 was produced as a separate product at a location different from that used for the other OM packages. In addition, the design of the system was largely created by external contract workers. Due to the geographical distance and the fact that PSS1

was implemented as a separate project, there were communication problems. Many issues with the PSS did not surface until integration with the other packages started. By that time, the documentation, including the design, was seriously obsolete which further hindered integration.
- *Process enforcement issues*. The internal evaluation of PSS1 revealed that the usual process for change requests had not been followed. Consequently, the PSS1 package had many undocumented changes. In addition, several changes were characterized as 'quick fixes'.
- *Experience levels*. The team assigned to the PSS1 development consisted largely of relatively inexperienced personnel. As a result, many errors were made that more experienced personnel would not have made. In combination with the already mentioned communication issues, this probably was the main cause for the quality problems.
- *Release pressure*. The PSS1 version could not be finished in time for the 5.0c release. However, the release pressure had caused the personnel to hurry and bypass the change request process. Consequently, the quality of the PSS1 was below Baan's quality standards. Because of this PSS1 was not made widely available to customers. For PSS2, better planning was done and the product was released on time and with the right quality.

4.3. Solutions

To address the identified issues, a number of measures were taken. The evaluation had shown both the functional design and the data model could be reused in PSS2 even though some modifications were required. The following requirements were specified for PSS2.

- *Integration*. Many of the problematic requirement changes in the PSS1 development concerned the integration with other Baan packages. Consequently, integration was made a priority in the development of PSS2.
- *Standardization*. The standardization of terminology and a more consistent use of this terminology in designs and source code were required.
- *DAL*. The use of DAL scripts to access the database were to be enforced.
- *User interface*. The user friendliness needed to be improved.
- *Refactoring*. The existing code needed to be restructured to improve the quality. About two thirds of the available time was reserved for this.

Based on these requirements, a plan was made for the realization of PSS2. To avoid problems with requirement changes, Boeing functional experts were involved during the development.

Traditionally, many features were implemented using copy–paste reuse. To create a new feature, existing code was duplicated and edited. The obvious disadvantage is that any bugs in the duplicated code will have to be fixed twice. To address this issue, templates were introduced. A template is a reusable piece of code, specifically intended to be copied and completed. A template can be seen as a way to emulate inheritance in a non-object-oriented language. While things such as object identity and polymorphism are not supported in the Baan 4GL language, the Baan templates allow for creating a skeleton of reusable code in an object-oriented fashion.

A substantial amount of time was spent on refactoring the various modules in the PSS package. Large procedures were split into smaller ones (making the code both more readable and more reusable). A lot of time was spent on making the code readable. The interviewees cited examples where huge case

statements were reduced to a handful of lines of code. In addition, a lot of so-called dead code was eliminated.

Finally, large modules were split into smaller, more cohesive modules. This has the advantages of improved concurrent working and increased understandability. First of all, smaller modules are easier to understand. The second advantage is that the use of smaller modules makes it easier to work with multiple team members on the system since, typically, a module is locked when it is checked out by a team member.

4.4. Analysis

The transition from PSS1 to PSS2 can be considered very successful. Currently, Boeing has used a version of Baan ERP that includes PSS2 for more than one and a half years without any serious issues. This version is not exactly the same version as the current version of iBaan ERP 6, which includes additional functionality not needed by Boeing.

PSS 1 and PSS 2 can be considered as two internal releases leading to, so far, the first release of PSS as a part of the iBaan ERP version 6 product line. Speaking of design erosion in the context of this release may seem odd. However, we think it is appropriate. During the development of PSS, both requirements and architecture evolved significantly. In this case, we have seen that even in one version of a software product, symptoms of design erosion may surface. Similar symptoms as in the QP case were identified. For example, symptoms included uncertainty about specifications, problems with respect to maintenance and various design issues.

Most of the problems mentioned in Section 4.2 have been addressed. By focusing on restructuring the code, the problems with respect to code quality have been addressed. Consequently, the defect rates have dropped dramatically. Interviewees estimated that they had fixed approximately three defects per KLOC since the first release of PSS2. During the realization of the PSS2, this number was much higher.

A few practices can be identified that were responsible for the reduced defect rate.

- More attention was paid to the quality of the code. Consequently, the overall quality of the system has increased substantially.
- Two thirds of the realization phase for PSS2 was reserved for restructuring and refactoring. This allowed personnel to carefully review the source code and eliminate bad solutions.
- The customer, Boeing, was closely involved in the realization phase. Baan personnel frequently had contact with the functional experts at Boeing. Consequently, there was continuous feedback on the quality and functionality of the system.
- There was more focus on the structure of the system. Particularly the introduction of templates and the enforced use of DAL scripts improved the structure of the system making the system both more flexible and maintainable.

During the realization phase of PSS2, only one significant issue was encountered (mainly due to time pressure). The integration with various other Baan packages proved to be much harder than was anticipated. Consequently, the original plan for the realization phase did not allocate enough time for this. Yet, the PSS2 package was released on schedule. Despite the integration problems, PSS2 was delivered to Boeing on time and relatively few defects have been reported in the subsequent maintenance of the system. Currently PSS2 is shipped with the generally available iBaan ERP version 6, which has been deployed on many customer sites.

5. DISCUSSION

In this section, we address the research questions raised in the introduction (Section 1) and present a number of recommended practices.

5.1. Research questions

In the introduction, we listed five research questions about the symptoms, identification, causes, resolution, and prevention of design erosion during software evolution. The two cases we presented in this paper are examples of systems where erosion was recognized and repaired. The first case is a good example of erosion over multiple versions of a software system whereas the second case illustrates how design erosion may affect and delay the development of a single version of a software system. In the following sections we address the research questions using the results of these case studies.

5.1.1. Symptoms: what are the effects of design erosion on a system?

A first step in preserving the design of a software system is to recognize the symptoms of a deteriorating system. Both of the cases we examined exhibited similar symptoms of deterioration.

- *Code quality*. In both cases, the personnel working with the system were unhappy with the quality of the source code. They felt that the quality of the source code had increasingly become a problem for working with the system. Quality problems included unnecessarily complex or lengthy functions, abuse of language features, wrong use of infrastructure features, etc.
- *Uncertainty about specifications*. There was a great deal of uncertainty about the specification of the system. The designs were sketchy and incomplete. In the case of the QP, the personnel actually depended on unspecified and even incorrect behavior of the system. In the case of the PSS, undocumented changes had been added to the system effectively making the existing design specifications obsolete.
- *Regressions*. In both cases, fixes for defects often introduced new problems. The PSS package, for instance, had seen a high number of defects fixed in the period prior to the release of the prototype. According to the interviewees, this indicated that the quality of those fixes was low and probably introduced additional problems. In the QP case, the near certainty of future regressions was an important motivation to completely redo the software in order to address the long list of known defects rather than to address each defect individually.
- *Deployment problems*. In both cases, there were problems with respect to the usage of the system. The PSS1 package was considered too problematic to be released to customers and was not made generally available in the Baan 5.0c release for which it was scheduled. The QP component was deployed and used successfully for several years. However, personnel had to learn to work around bugs in this component and were even depending on its incorrect behavior.
- *Defect rates*. In both cases, the interviewees indicated that prior to the rework the amount of defects that needed to be fixed was substantially higher than in comparable systems. In the QP case, the personnel also indicated that the average time needed to fix defects was much higher (up to a week per defect).

5.1.2. Identification: how does an organization decide that its software is eroding and needs to be repaired, and how does this decision process work?

In order to be able to decide what to do with an eroded system, it has to be recognized first that the system is eroded. Additionally it must be established whether it is worthwhile to undertake an effort to repair that software. In the systems we examined, Baan concluded that something needed to be done. A number of factors may play a role in identifying erosion.

- *Defect densities*. Baan uses a company-wide system to track the reporting of defects and their repair. Defect figures in a particular software component, which are larger than the norm, automatically trigger management to initiate proper action. In particular, in the QP case these metrics played an important role in the decision to redo this component.
- *Evaluation*. In both cases, the decision to evolve the system was taken after an internal evaluation of the software. In both cases these evaluations were prompted by problems with the existing software and a general feeling that the software was not in a good condition (e.g., because of the symptoms outlined above).
- *Additional requirements*. New requirements may call for enhancements that, given the quality of the system at that point, are infeasible. In the case of the QP, there were approximately 100 defects that needed to be fixed and, in addition, there was also a number of features that would likely be required in the future. This situation played an important role in the decision to redesign the QP component. The problems with the PSS package became apparent when integrating the PSS package with other packages (because of new customer requirements). This led to the internal evaluation that triggered the PSS2 project.
- *Change of staff*. Personnel working in information systems, like most human beings, may be reluctant to admit their own faults. In both cases, the personnel who identified the erosion and took the initiative to evolve the software had not been involved in the original development of the software. In the QP case this was because people had left Baan whereas in the PSS case, the work was assigned to different teams.

5.1.3. Causes: what are common causes for erosion?

In order to effectively repair an eroded system, the causes of the issues that are responsible for the erosion need to be understood. We have found that both cases had a number of common issues. Consequently, these issues are also likely to share the same causes.

- *Uncertainty about the evolution of the system*. In both cases we reconstructed the evolution of the involved systems by interviewing the involved personnel. We found that the persons we interviewed did not always agree on the details. For example, in the PSS case, some personnel attributed particular changes to wrong versions of the system. In the QP case on the other hand, one of the interviewees indicated that he had only been working on the NQP and never worked on the old QP components. Consequently, he could not tell us very much about the various versions of the old QP.
- *Lack of knowledge about early design decisions*. In both cases, all or most of the original developers were either no longer working on the system or had left the company entirely. Consequently, many of the design decisions taken early in the evolution of the software were

poorly understood. Particularly in the case of the QP, maintenance became more problematic after the person who had designed QP left Baan. In the case of the PSS package, the designers worked in a different location than the people who implemented it. After the transfer of the work to another center, the group of people assigned to work on the PSS system had not previously been involved with the PSS system.

- *Too little attention to design during evolution*. During the evolution of a system, changes may occur that require that the software design be altered. In both cases, we found that little attention to design was paid during the evolution. In particular, in the QP case there had been significant, mostly undocumented, changes to the design and even the architecture of the system. The cumulative effect of these changes defined the behavior of the system rather than that the software implemented a particular specification. This can partially be explained by the fact that during the lifetime of the system, software practice within Baan R&D evolved substantially. We mentioned earlier the increased rigor in software processes as well as the extensive standards for application programming in the Baan 4GL language.
- *Quick fixes*. During the evolution of a system, defects are found and fixed (in [12] this is called corrective maintenance). The proper way to fix a defect is to analyze the defect, design a solution, implement, and test the solution. Unfortunately, time-pressure or cost considerations may prevent the personnel from properly following this process. Often this results in quick fixes that address the issues but that may also introduce additional issues. In particular, in the PSS case we observed that the existing process for processing change requests (which is the common way for fixing large issues) had not always been followed. The QP components never had a proper design and consequently any changes to that design had never been properly documented.
- *Release pressure*. Both of the systems we examined are part of the Baan ERP product line, which evolves through a series of regular releases [12]. Work on the software in the systems in the product line has to be synchronized with these releases [12]. In the PSS case this proved to be an issue with the first release of PSS. The project for PSS2 included a careful planning phase and the software was finished in time for the release. In the QP case, the work on NQP was split over two releases to ensure that each release would include a working QP.
- *Requirement changes*. In both cases, new requirements were added which were problematic (also see identification in Section 5.1.2). In the PSS case, the initial requirements formed the basis for a prototype. However, during the work on this prototype, new requirements were added to the project. In the QP case on the other hand, the QP components went through several revisions that radically changed the behavior and functionality of the components.
- *Education*. In particular, in the PSS case a significant amount of the problems can be attributed to personnel inexperience. Had they been more aware of the processes and programming techniques in the Baan ERP product line, some of the problems could have been avoided.

5.1.4. *Resolution: what kinds of solutions are applied to fix an eroded system, and how and when do decisions need to be made with respect to preservation and repair?*

Once it has been determined that a system is eroded, and once causes have been identified, an attempt can be made to repair the system and prevent further damage. The obvious things that can be done and that we have observed in the case studies are as follows.

- *Redevelopment*. Redevelopment of the software is often the only real option during evolution for fixing an eroded portion of a system. This approach was chosen in the QP case since the evaluation showed that merely addressing the known defects would not result in a better quality system, which was required to serve as a base for additional enhancement.
- *Restructuring*. The people working on the PSS chose to restructure the existing system and reserved a significant amount of time for it.
- *Strong focus on design*. As pointed out earlier, the lack of up to date designs and specifications is usually one of the problems with eroded systems. In both cases, recovering/updating the designs was an integral part of the attempt to address the problems and a key to the success of the whole operation. By making the design more explicit, the QP personnel were able to test their software more effectively. In addition, unspecified/unintended behavior could be treated as a defect because the correct behavior was specified.
- *Modularization and object orientation*. In both cases, the personnel complained about the fact that the source code was in bad shape and that there were many dependencies between the various modules and components in the system. In both cases object-oriented type mechanisms such as encapsulation, information hiding, and delegation were applied to improve the structure of the system. In the PSS case this resulted in smaller modules, whereas in the QP case, the stage 1 development was able to reuse classes from the stage 0 development.

5.1.5. Prevention: what practices help prevent erosion and what were the good practices that were applied in both cases?

The maintainers of the systems we examined in this paper have experienced first hand what it takes to recover a deteriorated system as they evolve. Naturally, they made an effort to learn from their experience to adapt the ways used such that future problems can be avoided. In the cases we examined, a number of practices were adopted that appear to be successful in forestalling future problems.

- *Automatic regression testing*. In order to prevent new defects from being introduced during defect fixing, automated tests can be used to verify that the system still works satisfying requirements. In the QP case, automated tests were created for each new feature. In addition, when defects were fixed, an automated test was created to verify that the defect was fixed. Over time, the number of automated tests has grown to more than 800 tests, which are run after each change to the system. Personnel are now confident that if the system passes all tests after a change that no regressions have occurred.
- *No undocumented fixes*. Both the QP and the PSS shared the problem that in the past there had been undocumented changes. This makes it hard both to test the system and to use the system correctly. To address this, a detailed process for fixing defects is now commonly used in the PSS system. In the QP system, any behavior that is not specified is considered to be a defect and the proper way to correct or add new behavior starts with a proper design phase.
- *Stronger focus on process*. Part of the problems with the PSS can be attributed to the fact that the existing software work processes were not enforced. It has been good practice to follow a certain process for incorporating customer change requests, for instance. In the old PSS, time pressure often caused personnel to by-pass this process. In the QP case, the processes evolved during its evolution. The NQP evolution was done using the same processes that are used throughout Baan nowadays.

- *Product releases*. Software work within Baan is driven by the product releases. Consequently, any adaptation projects such as the adaptive, corrective and enhancive projects of both cases, need to be finished in time in order to be ready for the release. This requires careful planning because generally, product releases are not delayed because individual projects are not ready. In the QP case, this was the reason to plan a two-staged release. In the PSS case, one of the problems was that the initial release was not finished in time for a product release.
- *Team composition*. A problem in the PSS case was that the software work was done by a relatively inexperienced team. A consequence of this was that errors with respect to process and technology were made that would likely have been avoided by a more experienced team.

5.2. Good practices

We highlight a few of the practices we observed since we believe that they are good practices, and that others who are involved in software evolution could benefit by taking notice of them as well.

5.2.1. Diagnosis of design erosion

As argued earlier, in both cases the interviewees cited a number of symptoms that they observed in their systems. By being alert to these symptoms, organizations may be able to identify problematic subsystems in an earlier stage and act appropriately. Some good design erosion diagnosis practices we observed are as follows.

- *Monitoring defect metrics*. Important symptoms of design erosion are the defect rate and the average time needed to fix these defects. Defect rate metrics are relatively easy to collect, and can serve as an early warning system for problematic systems. The number of defects is relevant, but so is the average cost/time to fix a defect. In both cases, these two metrics showed that something had to be done. Monitoring of these metrics by management may help identify problems with respect to eroding designs at an early stage. The defect reporting system also assists in discovering the precise location of the defects, even in case of defects having roots in or ripples into other components.
- *Performing code reviews*. Code reviews and Fagan inspections may help to detect problems with respect to code quality and design problems [14]. In both cases, personnel were complaining about the poor quality of the implementation code. They identified this as an important reason as to why it was hard to make changes to the systems.
- *Involving different people*. An important aspect of both cases was that the decision to replace/repair the system was not taken by the original developers of the system, but by people who had inherited the system from their predecessors. Apparently, it helps to involve new people in diagnosing an eroding system and in evolving it to improve it.

5.2.2. Decisions about the future of the system

As became clear in the Baan cases, once it has been determined that a particular subsystem is eroded, we observed three things done to evolve the system.

- *Repair the system*. In the PSS case, the personnel decided (after an evaluation) that the system could be repaired and that replacing it would be more expensive than repairing it. So a plan was made to repair it.
- *Replace the system*. In the QP case, the affected personnel concluded that replacing the system was economically more feasible than attempting to fix the known defects. A two-staged replacement scenario was chosen.
- *Nothing/continue to maintain system*. Sometimes it may not be economically feasible to either repair or replace the system. In such cases, the choice may be made to do nothing at all. This may mean, for instance, that the organization accepts that change requests for such a system have a high cost or that the operational dependencies on the system are reduced.

We observed that the decision as to what to do with a system should be based on a thorough evaluation of the situation and should include expectations about future requirements and change requests, estimates of maintenance/evolution costs, estimates on additional license revenues etc. It should be noted that doing nothing also has an associated cost (e.g., increasing corrective maintenance cost, reduced ability to absorb new requirements, etc.). Interestingly, in both cases the evaluation was done by people who were not involved in the initial development. This suggests that these kinds of evaluations should preferably be done by people who are not directly involved with the doing the work.

In particular, the approach taken in the QP case seems illustrative. There the personnel simply took the number of existing change requests, the knowledge that each change request had a certain associated cost (which was much higher than it should be), and multiplied the numbers, which gave them a rough estimate of what it would take to get the system in an acceptable state. Then they also took into account that it was very likely that this maintenance effort would introduce new, equally hard to fix defects and that there would likely be more change requests by the time they had finished. Based on this assessment, it became apparent that completely replacing the QP existing system was more cost-effective as a way to achieve the evolution in the product line than continuing to maintain that QP system.

A process that might be suggested is to be alert to symptoms of design erosion. In case of strong indications that something is wrong, an evaluation such as outlined above should take place. Preferably, external people should be involved in the evaluation to allow for a more objective judgment. Based on a cost estimation of the three outlined measures, a decision may then be made.

5.2.3. Solutions

In both cases, we have seen that a number of technical and non-technical practices have been adopted to resolve the situation and prevent further damage.

- *Object orientation*. In particular, in the QP case the personnel used object-oriented mechanisms to their advantage to improve modularity, extensibility, and reusability. However, in the PSS case, personnel also applied typical object-oriented mechanisms such as information hiding and even a primitive form of inheritance to reduce the number of dependencies and reduce the amount of copy-paste reuse that plagued the old version of the software.
- *Process*. In an organization as large as Baan R&D, process is the key means to control the progress of the software work. In the PSS case, there were a number of issues with the

enforcement of existing processes. This led to a situation where qualitatively poor enhancements slipped through and where specifications and designs no longer accurately represented the functionality of the system. The PSS case in particular benefitted from a stricter enforcement of the existing processes, which are designed to prevent many of the problems the original PSS system had.

- *Regression testing*. While automated tests cannot completely ensure that a system is free of bugs, it does help ascertain that new changes do not break existing functionality. In particular, the QP case demonstrated that creating automated tests for each feature and each change request helps prevent regressions and increases the overall stability of the system.
- *Refactoring*. In the PSS case, a significant amount of time was reserved for refactoring. Convincing the management of the necessity to reserve time for refactoring proved hard. However, the success of the refactoring in the PSS case has clearly demonstrated the benefits.
- *Release planning*. Because large maintenance efforts may not fit within scheduled releases, Baan has implemented a company-wide requirements management process to estimate and schedule the different requirements over the subsequent releases [6]. Large maintenance efforts are split into several requirements, which are mutually related, and for each of them the workload is estimated by the architects. In the QP case this process led to the decision to divide the work into a stage 0 and a stage 1 release. Both of these stages were completed and released in time to be incorporated in a major Baan release.

5.3. Related work

Perry and Wolf, in their paper on software architecture, make a distinction between architecture erosion and architectural drift [4]. Architectural erosion, according to Perry and Wolf, is the result of 'violations of the architecture'. Architectural drift, on the other hand, is the result of 'insensitivity to the architecture' (the architecturally implied rules are not clear to the software engineers who work with it). The second case study is an example of architectural drift since the understanding of the system decreased by assigning the work to a new team.

Parnas, in his article on software aging [3], observes similar phenomena. Although he does not explicitly talk about erosion, he does talk about aging of software as the result of bad design decisions, which in turn are the result of poorly understood systems. In other words, erosion is caused by architectural drift. As a solution to the problem, Parnas suggests that software engineers should design for change, and should pay more attention to documentation and design review processes. He also claims that no coding should start before a proper design has been delivered. In [2], Jaktman *et al*. present a set of characteristics of architecture erosion. Some of these characteristics are also identified in our own case study. In their case study, Jaktman *et al*. aimed to gain knowledge about how architecture quality can be assessed. Assessing architecture erosion is an integral part of this assessment. Finally, in [15], Glass discussed how accumulated changes may push a product to the 'ragged edge of its design envelope', where small changes tend to be very expensive.

To avoid making bad design decisions, software personnel can consult a growing collection of patterns (e.g., the GoF (Gang of Four) book [16] and Buschmann *et al*.'s collection of architecture styles and patterns [17]). However, while design patterns are good for solving specific design problems, a side effect of applying them is usually that the design becomes more complicated.

A relatively new approach to countering design erosion is refactoring [18], a process where the existing implementation of a system is changed to improve the design of the implementation. Fowler *et al*. presented a set of refactoring techniques that can be applied to a working program. By using these techniques, some violations of good design can be replaced. Unfortunately, some of the refactoring techniques can be labor intensive, even with proper tool support (e.g., Roberts *et al*. [19]).

Yet another approach is to pursue separation of concerns. The general idea behind separation of concerns is to capture and modularize the functionality related to a particular concern. By separating concerns, the effect of changes to a concern can be isolated. For instance, by separating the concern about synchronization from the rest of the system, changes in the synchronization code will not affect the rest of the system. Examples of experimental approaches that aim to improve separation of concerns are Aspect Oriented Programming [20], Subject Oriented Programming [21] and Multi-Dimensional Separation of Concerns [22].

However, as is demonstrated in this paper, design erosion is as much a result of non-technical issues (e.g., the work processes) as it is from purely technical issues. In fact, a significant amount of the measures that were taken in the Baan cases can be characterized as non-technical. It is not surprising that many of these measures are derived from or inspired by recent research into software development and maintenance methodologies. For example, the automated tests in the QP case that are used to verify whether the system has regressed or not and still meets its requirements is also a center piece of many spiral and iterative methods for software evolution (e.g., [23–25]).

Glass, in his book on software engineering facts and fallacies, grudgingly admits to non-technical factors such as management being more important than technical factors in both the successes and failures of software projects [15]. Our case studies confirm this.

5.4. Future work

As we indicated in Section 2, our research approach is based on mostly qualitative data. There are a number of reasons we provide only a limited amount of quantitative data (e.g., in the form of qualitative statements about internally available defect metrics). The most important reason is that such quantitative data is considered to be sensitive data in the software industry. In addition, due to the explorative nature of the study, there are no predefined hypotheses that can be easily tested in a quantitative fashion. Our study, however, can give rise to additional quantitative research. For example, the relation between defect metrics and software quality could be examined in more detail.

As indicated in Section 2.4, Baan is a large software company, which can be seen as representative for the domain they operate in. However, we believe that some of our conclusions are more widely applicable. Therefore, it would be worthwhile to repeat this study across a number of other product software companies to identify additional best practices and to validate whether the findings of this study can be generalized to other companies within the same domain and software companies working in different domains.

6. LESSONS LEARNED

The most important lesson that may be learned from both cases is that design erosion is not simply the result of incompetence, negligence, or other failures of either organization or individuals.

Instead, it is the result of the ever-changing context of the system. Over time, staff, organization, processes, requirements, and technology are all subject to change. These changes affect the evolution of the software and all contribute to a gradual degradation in software quality.

As argued in our earlier research [5], there is a certain inevitability to design erosion. Therefore, a software organization should not be judged by whether or not parts of its software are eroded but by *how* they deal with the situation. A properly functioning software organization will be able to detect design erosion, and based on a thorough analysis of the situation will also be able to determine whether it is economically feasible and desirable to address it. Judging from our two case studies, Baan R&D is a healthy organization in that respect since it properly identified and repaired eroded software sub-systems in their product in a controlled style. Both sub-systems are in a satisfactory condition now and their respective maintainers have adjusted their work processes in order to prevent further problems. Throughout both projects, Baan shipped multiple, fully-functional Baan ERP releases to customers. In the QP case, a working QP component was shipped each time whereas in the PSS case, it was decided not to make the prototype available to customers.

The case study also confirms some of our conclusions from our earlier work [5]. In particular, the QP case has demonstrated how subsequent releases of a software product may introduce design changes that conflict with earlier decisions. These changes arise from new or changed requirements that, in earlier releases, were not anticipated or foreseen. The PSS case is less illustrative in this respect, but it does clearly demonstrate that design erosion can occur within one release given enough changes in the software's context. Additionally, it is illustrative of the fact that non-technical factors play at least as large a role as technical factors. For example, assigning the maintenance of the PSS system to another team caused problems with respect to the understandability of the software. In addition, process issues were an issue with the PSS case.

The main purpose of these exploratory case studies was to investigate problems and issues with respect to design erosion in software developing organizations. By selecting two systems that we knew had recently gone through extensive refactoring to address significant problems in these systems, we have been able to reconstruct the evolution of these systems and analyze how the design erosion was identified, what the causes were for the problems, and how they were compensated for.

In both cases, before addressing the erosion problems, an internal evaluation was done to analyze the system. This evaluation was triggered by a number of symptoms, including code quality, deployment, and other problems. The defect metrics that are routinely collected by Baan played a crucial role in both cases. Another important factor was the fact that the people who initiated the evaluation had not been involved with the initial development of the involved software.

Altogether, the design erosion was caused by a combination of technical (e.g., the QP design was problematic) and non-technical factors (e.g., in the PSS case the change process was not properly enforced). In particular, the QP case confirms some of the causes for design erosion we identified in [5]. Subsequent design decisions essentially reversed some of the design decisions taken in the beginning of the evolution of this software component. In the end, it proved easier to redo the component to meet the new requirements rather than continue to maintain the result of the accumulation of more than a decade of adaptive and corrective maintenance. Most of the adaptations we identified arose from new requirements that would have been hard to predict when the original QP component was designed (e.g., supporting SQL features that did not even exist when the QP component was first developed).

This confirms our most important conclusion from [5]: design erosion is inevitable because subsequent, unpredictable changes may be incompatible with earlier design decisions. As we have seen in both cases, it can be cost effective to do something about it. In the case of the QP component, it proved to be cost effective to invest in a replacement component, whereas in the PSS case an extensive refactoring project addressed the identified problems adequately.

In addition to analyzing problems and causes, we examined the way the problems are solved. The solutions included the adoption of processes of higher professional standards, object orientation, requirements management processes, refactoring, and regression testing. The applied solutions also confirm that design erosion is a serious problem since both software systems required expensive evolution projects to address the erosion problems. However, it was determined beforehand that these projects would be cost effective, and both projects were taken to very successful conclusions.

ACKNOWLEDGEMENTS

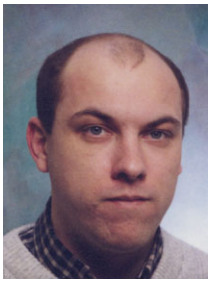
We thank SSA Global–Baan for its cooperation in these case studies. In particular, we thank Gerwin Ligtenberg, Bart Kasteel, Peter Romeijn, and Pierre Breuls for the discussions about the two case studies and their feedback on this paper.

REFERENCES

1. Cusumano MA, Yoffie DB. *Competing on Internet Time: Lessons from Netscape and its Battle with Microsoft*. The Free Press: New York NY, 1998; 384.
2. Jaktman CB, Leaney J, Liu M. Structural analysis of the software architecture—a maintenance assessment case study. *Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA1)*. Kluwer: Deventer, The Netherlands, 1999; 455–470.
3. Parnas DL. Software aging. *Proceedings of the International Conference on Software Engineering (ICSE 1994)*. ACM Press: New York NY, 1994; 279–287.
4. Perry DE, Wolf AL. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes* 1992; **17**(4):40–50.
5. Van Gurp J, Bosch J. Design erosion: Problems and causes. *Journal of Systems and Software* 2002; **61**(2):105–119.
6. Svahnberg M, Bosch J. Characterizing evolution in product-line architectures. *Proceedings of the IASTED 3rd International Conference on Software Engineering and Applications*. Acta Press: Calgary, Canada, 1999; 92–97.
7. Brinkkemper S. Method engineering with Web-enabled methods. *Information Systems Engineering: State of the Art and Research Themes* Brinkkemper S, Lindencrona E, Sølvberg A (eds.). Springer: London, 2000; 123–133.
8. Basili V. Editorial. *Journal of Empirical Software Engineering* 1996; **1**(2):1–2.
9. Basili V, McGarry FE, Pajerski R, Zelkowitz MV. Lessons learned from 25 years of process improvement: The rise and fall of the NASA Software Engineering Lab. *Proceedings of the International Conference on Software Engineering (ICSE 2002)*. ACM Press: New York NY, 2002; 69–79.
10. Seaman CB. Qualitative methods in empirical studies of software engineering. *IEEE Transactions of Software Engineering* 1999; **25**(4):557–572.
11. Elmasri E, Navathe SB. *Fundamentals of Database Systems*. Benjamin/Cummins: Redwood City CA, 1994; 600.
12. Chapin N, Hale JE, Khan KMd, Ramil JE, Tan W-G. Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* 2001; **13**(1):3–30.
13. Brinkkemper S. RE for ERP: Requirements management for the development of packaged software. *Proceedings 4th International Symposium on Requirements Engineering (RE'99)*. IEEE Computer Society Press: Los Alamitos CA, 1999; 159.
14. Van Genuchten M, Van Dijk C, Scholten H, Vogel D. Using group support systems for software inspections. *IEEE Software* 2001; **18**(3):60–65.
15. Glass RL. *Facts and Fallacies of Software Engineering*. Addison-Wesley: Reading MA, 2002; 224.
16. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading MA, 1995; 416.

17. Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley: New York NY, 1996; 476.
18. Fowler M, Beck K, Brant J, Opdyke W, Roberts D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley: Reading MA, 1999; 431.
19. Roberts D, Brant J, Johnson R. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems* 1997; **3**(4):253–263.
20. Kiczalez G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J. Aspect oriented programming. *Proceedings of ECOOP 1997*. Springer: Berlin, 1997; 220–242.
21. Harrison W, Ossher H. Subject-oriented programming (a critique of pure objects). *Proceedings of OOPSLA 1993*. ACM Press: New York NY, 1993; 411–428.
22. Tarr P, Ossher H, Harrison W. N degrees of separation: Multi-dimensional separation of concerns. *Proceedings International Conference on Software Engineering (ICSE 1999)*. IEEE Computer Society Press: Los Alamitos CA, 1999; 107–119.
23. Boehm B. A spiral model of software development and enhancement. *IEEE Computer* 1988; **21**(5):61–72.
24. Beck K. *Extreme Programming Explained*. Addison-Wesley: Boston MA, 2000; 190.
25. Cockburn A. *Agile Software Development*. Addison-Wesley: Boston MA, 2001; 256.

AUTHORS' BIOGRAPHIES



Jilles van Gurp is a software architect and engineer at Creative Development, a manufacturer of a content management software product in Nijmegen, The Netherlands. He obtained a computer science masters degree from the University of Utrecht in 1998, a licentiate degree from the Blekinge Institute of Technology in Sweden in 2001 and a PhD from the University of Groningen in 2003. In 2003 he worked at the same university and continued research in the context of the IST Status project and the ESF RELEASE network.

Sjaak Brinkkemper is professor of Organisation and Information at the Institute of Information and Computing Sciences of the University Utrecht, The Netherlands. Previously, he was a consultant at the Vanenburg Group and a Chief Architect at Baan. Before Baan, he held academic positions at the University of Twente and the University of Nijmegen, both in The Netherlands. His research interests include software product development, information systems methodology, meta-modelling, and methods engineering. He holds a BSc from the University of Amsterdam, and a MSc and a PhD from the University of Nijmegen.

Jan Bosch is a vice president and head of the Software and Application Technologies Laboratory at Nokia Research Center, Finland. Earlier, he headed the software engineering research group at the University of Groningen, The Netherlands. His research activities include software architecture design, software product families, software variability management, and component-oriented programming. He has organized numerous workshops, and served on many programme committees and steering groups. He received a MSc degree from the University of Twente, The Netherlands, and a PhD degree from Lund University, Sweden.

