Decoupling Functionality To Facilitate Controlled Growth

Roelof J. van den Berg and Arian J.R. Zwegers
Department of Technology Management
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven
THE NETHERLANDS
e-mail: rbe@tm.tue.nl

Abstract: This paper describes the results of the action research project ' Gordian', which studied some measures to be taken in order to facilitate controlled growth of a software product. Technical decoupling prevents the consequences of three identified types of integrations, and it streamlines the integrations between packages. Conceptual decoupling prevents violations of the integrity of the data. Since the outcome of technical and conceptual decoupling is a static picture of the system, the dynamic element of the development process is addressed by means of organisational measures.

Keywords: software development, decoupling, controlled growth, Gordian project

Roelof J. van den Berg is a research co-ordinator with BaaN Company. He holds a MSc. degree cum laudae from Eindhoven University of Technology. This summer, he will receive his Ph.D from the same institution. His research interests include the design of information infrastructures and modular software design.

Arian J. R. Zwegers received his MSc. degree cum laudae from Eindhoven University of Technology in 1993. Currently, he is preparing his Ph.D thesis at the same university. In 1996 , he was involved in the Gordian project at BaaN Company. His research interests include shop floor control architectures and enterprise reference architectures.

1. Introduction

To speak of life cycles in relation to information systems is far from being original these days. To draw analogies between the existence of information systems, and organic life has become such a cliché that one almost stops noticing how remarkably unjustified it still is. The way the average information system struggles on between its release and final shut-down differs from real life in so many respects, that one paper cannot cover even the most noticeable ones. For this reason, we decided to devote this contribution to only one of the chasms between real life and its counterpart in information systems: controlled growth, including decay, along predictable lines. This characteristic stands as a phenomenal challenge to the manufacturing system discipline. In this paper, we describe the results of an action research project, the Gordian project, concerning this facilitation of controlled growth through modular design.

The project was carried out at the BaaN Company, currently the largest software manufacturer in the Netherlands. The BaaN Company is a supplier of a standard software package for enterprise resources planning. Previously, the package was called 'Triton' ; currently, it is named after the company. The current version, BaaN IV, consists of eight packages (with e.g. Distribution, Manufacturing, Service and Finance functionality) and comprises more than 2.5 million lines of code.

During the fast growth of Triton, problems have emerged concerning the integrations between its various packages. Triton was introduced six years ago. It has grown fast ever since, and will continue to grow fast. Pieces were built on top of each other, and sometimes relations were created in a rather haphazard way. As a consequence, customers come across more problems, application developers solve more bugs, at the same time introducing more and more problems (bad fixes). Furthermore, whenever a new piece of functionality is introduced into Triton, congestions occur at subsequent development, documentation, and testing. Although exceptions have occurred, all packages have to be ready and are released at the same moment; spreading of workload is not possible. In other words, there are no buffers in between the various packages. The interfaces between packages could have been defined better. Although integrations are needed, it is desirable to decouple the packages, thereby decreasing the necessity to release packages at the same moment. New functionality is hard to add, and application development becomes a process that is difficult to control. In the remainder of this document, this entanglement problem is called the Gordian problem.

This paper is organised as follows. In the next section, the distinction in three aspects of resolution, namely technical decoupling, conceptual decoupling, and organisational embedding is explained. After this, we present the first aspect: technical decoupling. Then, the conceptual decoupling theory is introduced. Furthermore, we illustrate how organisational measures are needed in order to facilitate a controlled development process. We conclude this paper with a discussion on our findings.

2. Three Aspects of Resolution

In order to tackle the Gordian problem, we think we need to address the problem at a number of aspects. In order to put the packages on the market independently of each other, they need to be 'technically' (or physically) decoupled. However, in an ideal situation the technical decoupling follows a pre-determined architecture. This architecture is constituted of conceptually decoupled modules, indicates where the interfaces are between modules, and what these interfaces consist of. Finally, in order to prevent the architecture of being a one-time snapshot, organisational measures have to be taken which contribute to a controlled development process and a lasting software product.

Therefore, we distinguish three aspects of resolution:

- Technical decoupling;
- Conceptual decoupling;
- Organisational embedding.

Figure 1 illustrates the difference between 'technical' and 'conceptual' decoupling. For an elaboration of technical and conceptual decoupling, we refer to the next Sections.

3. Technical Decoupling

3.1 Types of Integrations

At a technical level, the Gordian problem occurs in a number of ways. In this paper, we focus on three types of integrations: table-table integrations, software-table integrations, and software-software integrations. These types of integrations are illustrated in Figure 2. Note that we focus on entwining of Triton packages, rather than Triton modules or even business objects. Packages contain modules, which in their turn consist of business objects. Packages are easier to decouple than modules, since they have their own type declarations and help texts.

The first type of integration is the table-table integration. References are made from a table in a certain package to a table in another package. Normally, a user does not notice these references, but he might run into them when he tries to delete a record of e.g. the main item table. Although sometimes items are not used anymore, they cannot be deleted, since references are made to these items. Figure 2 illustrates an example of a reference to an item record (table B) from a History by Item table in the Triton Process package. This table contains an item field, 'mitm' that refers to the item table. Deletion of the parent is restricted if any child refers to the parent. Although the item might not be produced anymore, and history data are obsolete, references still exist, and the item has to be present.

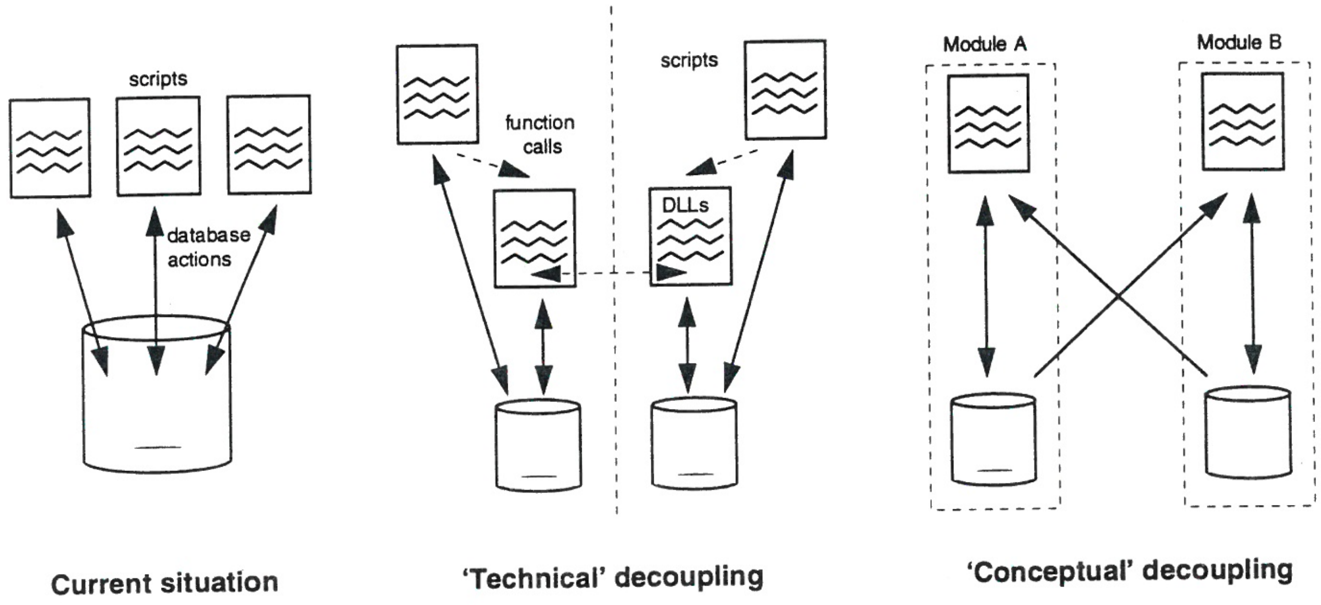

Figure 1. Decoupling

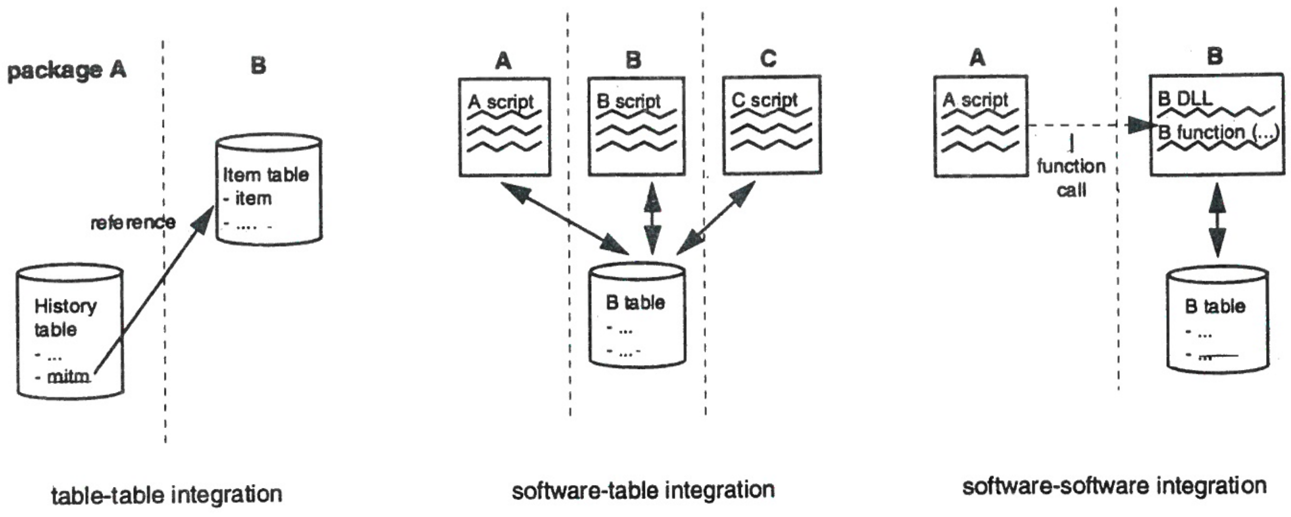

Figure 2. References Between Packages

References between packages imply that a customer needs to purchase the package whose tables are referred to. Consider the previous example in relation to a customer whose line of business is the process industry. If this customer has bought the Triton Process package, he needs to purchase the ITM module of the discrete manufacturing oriented Triton Manufacturing package as well, in order to have item control functionality. Clearly, decoupling the reference in question would be a contribution to avoiding this undesirable situation.

The second type of integration is the software-table integration. Scripts in a certain package write (and read) in tables of another package. Figure 2 gives an example, in which records of a package B table are deleted, inserted, or updated by scripts and Dynamic Link Libraries (DLLs) from other packages. Obviously, this kind of structure increases the entanglement between packages.

The third type of integration is the software-software integration. Scripts in a certain package call functions of a DLL in another package. An example is depicted by Figure 2, where a package A script calls a function of a DLL belonging to package B. Note that the particular script needs to 'know' the DLL mentioned. The script needs information about the package the DLL belongs to, the version of the package, etc. In addition, the script must check whether the DLL is present at all; it is possible that the user has not bought the specific module. In this kind of structure, all information for a proper invocation of other functions needs to be present in the script.

Besides the three types of integrations described above, we can distinguish some other types, which we do not discuss in this paper. Integrations occur at the domain level. A domain of an attribute is a type declaration. Integrations occur by referring to domains that are declared in other packages. Another type of integration is the integration at form level. Within a certain session, the user can zoom to other sessions. These other sessions do not necessarily belong to the same package as the session from which they are invoked.

3.2 Decoupling Integrations

In order to illustrate how integrations could be decoupled, we take a table-table integration as an example. Table B contains two fields: a key field called 'code' and an accompanying description 'desc'. A package A table refers to the 'code' field of the package B table. This reference could be resolved in two ways that are indicated in Figure 3. In the first solution (Figure 3, bottom left), the reference is deleted, and a field that gives the actual description is inserted instead of a field 'code' that indicates the *code* for the description. However, this option is not preferred, since consistency between records with the same description is not assured. Records that need the same description, might have differences in the description field.

The other option (Figure 3, bottom right), is to decouple the two tables - and thus the two packages - by using DLLs. In the current situation, deletion of the parent (a table B record) is prohibited if any child (any table A record) refers to the parent; when the parent is deleted, all references must be checked. A software solution that takes this into account would use two DLLs: one related to package A and serving as the interface for package B, and another one that is related to package B and serves as the interface for package A.

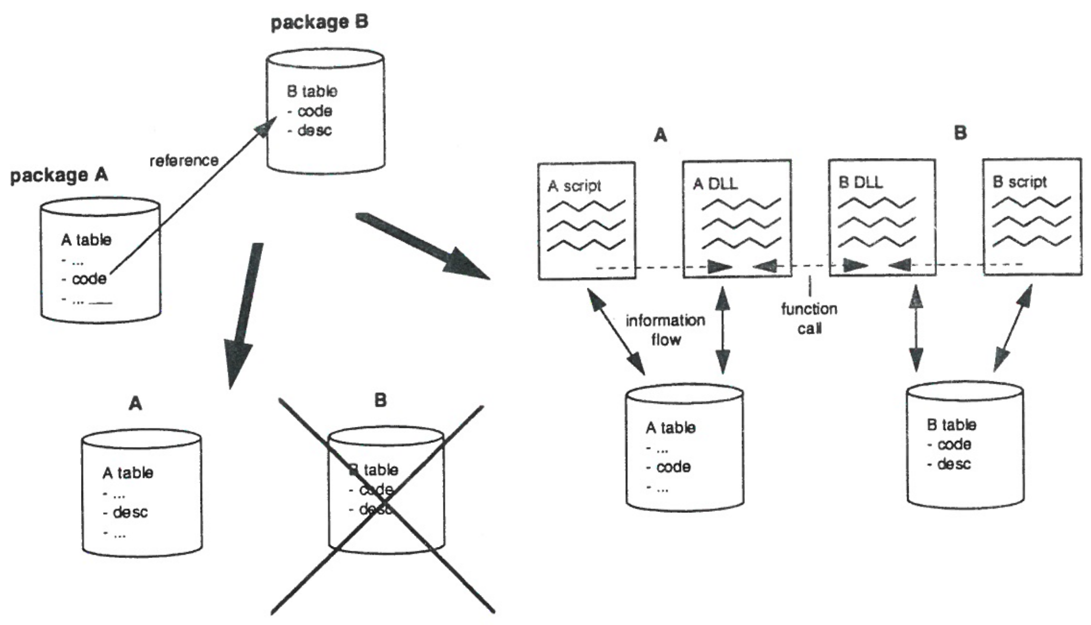

Figure 3. Two Possible Decoupling Options

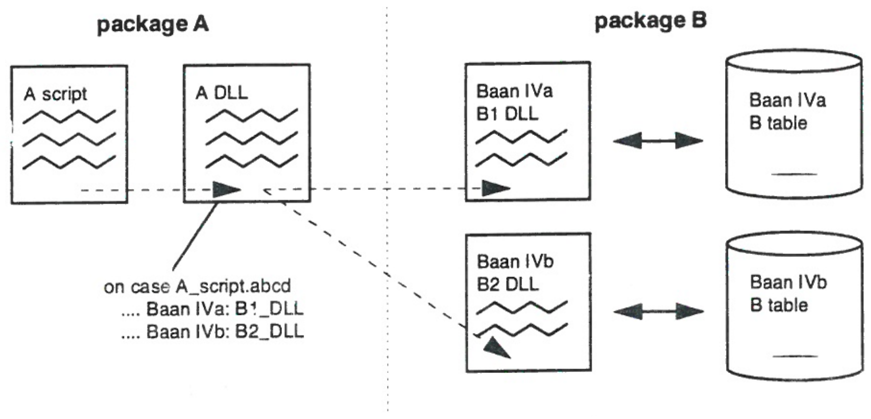

Figure 4. Parameter Dependent Invocation of Versions

The latter DLL should provide functions that return the description for a specific code. The first DLL should provide functions that check whether records are present with a specific code. These functions should be invoked when the user deletes a code.

In addition, there should be a provision (a parameter) that allows a user to invoke sessions such as those in a package A script depending on the implemented package B. In addition, a special solution is introduced for the situation that no package B is available. This so-called "minimal environment" could for instance consist of some primary tables. Figure 4 shows that a parameter determines the functions to be invoked in which DLL, depending on the implemented Triton and BaaN versions.

A double DLL has certain advantages over a single DLL. Note that a double DLL is used, where a single call from the A script to the B1 DLL would suffice. However, using a double DLL gives the advantage that the A script does not need to know all DLLs in other packages; it only has information about its 'accompanying' DLL. That DLL takes care of the proper invocation of other functions, whether they belong to package A, another BaaN Package, another BaaN version or even another vendor's software.

In conclusion, this Section illustrates the technical decoupling of a table-table integration where the referential integrity checking functionality of a DBMS is replaced by DLL functions. The former table-table integrations, i.e. the references between tables of different packages, are deleted. References between tables of the same package or references to common tables are still allowed.

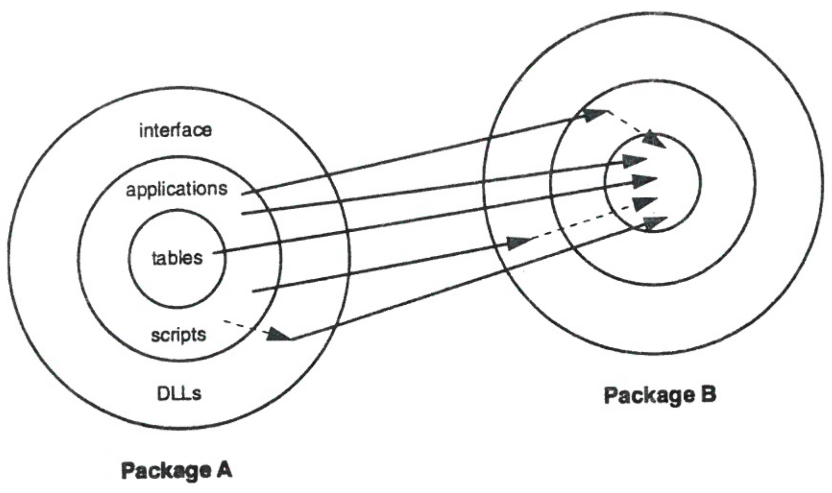

Figure 5. Current Situation

By introducing a special solution for the situation that no package B is available, two packages can be fully decoupled concerning the table-table integrations.

The approach applied in this Section may be carried out for all three types of integrations. Therefore, the other types of integrations are not described in detail in this paper.Consider the current integrations between two packages as illustrated in Figure 5. At the moment, there are five kinds of integrations, among which a few subtypes of the previously identified three main types (from top to bottom):

- an *include* of package B is included in a script of package A. This is one type of software- software integration.
- a script or *include* of package A writes (and reads) directly into a table of package B. This is a form of the software-table integration.
- a reference is made from a package A table to a field of a package B table. We call this integration the table-table integration.
- an A script calls a DLL function of package B. This is another type of software-software integration.
- a DLL of package A writes (and reads) directly into a table of package B. This is another form of the software-table integration.

After removing the five harmful types of integrations as depicted in Figure 5, a new, improved situation occurs. Figure 6 shows two packages where all table-table, software-table, and software-software integrations have been replaced by double DLL constructions. Note that a double DLL creation is also a kind of software-software integration, but it is 'technically' much more flexible than the two versions depicted in Figure 5 . For instance, compared to the mentioned integration types, a

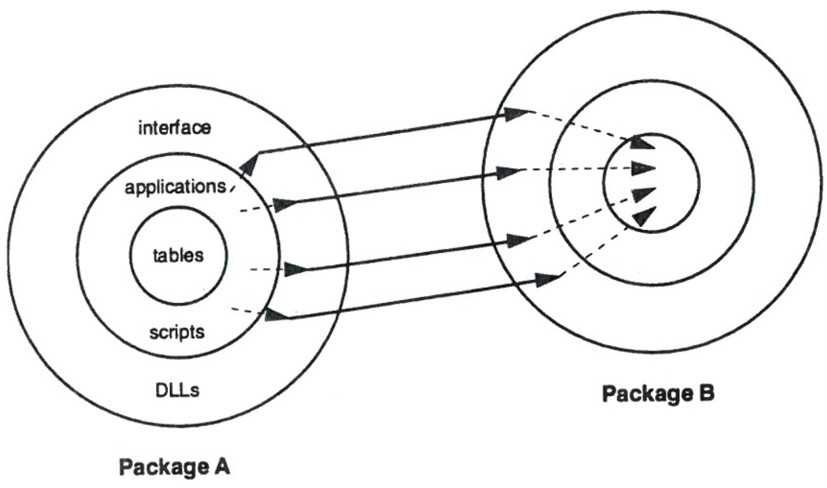

Figure 6. Ideal Technical Situation

double DLL construction is easy to adapt or extend.

4. Conceptual Decoupling

Conceptual decoupling of packages is needed in addition to technical decoupling. Adequate technical decoupling can significantly improve system maintenance, because it streamlines the integrations between the packages. However, it cannot prevent that one table is updated from various packages, which threatens the integrity of the data. Conceptual decoupling, through modular design, can solve this problem. Modular system design consists of the following steps:

- define the functions of the system,
- assign each function to a module,
- specify in detail the interfaces between the modules,
- design the non-interface part for each module separately.

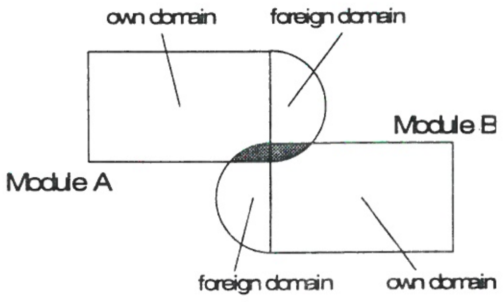

Figure 7. Modules and Their Domains

The principles of modular design have been formalised with respect to data structures in information systems in (Pels, 1988; Pels and Wortmann, 1990).These studies show the essential property of a module being that its interfaces are precisely defined and that a clear distinction is made between the input interface and the output interface.

Pels distinguishes a number of domains, which are illustrated in Figure 7. The own domain of a module contains the objects for which the module has retrieval and update authorisation (delete, insert, update). A further refinement can be made by distinguishing a private domain and a public domain. The private domain contains the objects of the own domain that are not visible to other modules, and the public domain comprises the objects of the own domain that are visible to one or more other modules. In Figure 7, the public domain of module A that is visible for module B and v.v. is in grey. Note that the public domain is the output interface, since it contains non-hidden information of the module itself.

The foreign domain of a module contains the objects for which the module has retrieval authorisation but no update authorisation. The foreign domain refers to the objects a module retrieves from other modules, and forms the input interface. It contains all specifications of other modules that must be known to design, validate and operate the module.

The union of the own and foreign domain is called the view domain. The view domain of a module contains the objects that are visible for the module. Being visible means that the objects are included in the information base of the module, and that these objects can be retrieved from this information base by the 'read' operation.

As made clear in the previous Section, an important part of the Gordian problem lies in the fact that tables in BaaN IV can be updated from various places in the system. Pels' theory indicates how this can be improved. Each table should be linked to one own domain only; update authority for a table should be located at only one place in the system. This last conclusion may not sound very original, but Pels provides a sophisticated approach to assigning update authority. He relies heavily on evaluating the constraints which each module assumes about a specific table. Communication clashes result if a module initiates an update that is in conflict with the actual value of the own data of another module. What combinations of values are meaningful and therefore allowed in the information base, is specified by means of constraints.Therefore, constraints play an important role in the cause and prevention of communication clashes.

However, problems in finding the constraints complicate application of Pels' ideas. According to Pels, modular decomposition makes it possible to develop and maintain an integrated information system for arbitrary large organisations. This is probably true, as long as their information systems are small because in practice the requirement to gather the constraints that are applicable to the identified modules hinders the suitability of his theory.

A crucial assumption is that all constraints are available in documentation (e.g. functional design). In practice, this assumption does not hold and most constraints can only be traced in the code. This limitation is of more importance to the applicability of Pels' framework than the number of constraints in a large system such as BaaN IV. The latter will only make the application of Pels' approach more time consuming. However, as long as constraints are hidden, application is impossible. Until documentation of constraints is significantly more developed, conceptual decoupling will be very fastidious.

5. Organisational Embedding

5.1 Introduction

In the previous Sections, we have discussed technical and conceptual decoupling. These two exercises are important, but not sufficient to sustain controlled growth of an information system. After all, the outcome of both technical and conceptual decoupling is only a snapshot of the system. They both emphasise the static element in system development. If done well, it certainly will make structured development of the system in the future more likely, but it provides far from a guarantee that this will indeed happen. For this purpose, the dynamic element of development should be addressed as well. This implies initiatives to increase the maturity of the development organisation.

Initiatives within the Capability Maturity Model framework (Humphrey, 1990) have already been taken within BaaN Development. Indeed many problems related to lack of controlled system growth can be solved through advancement in CMM. Along the same lines, a more individual approach can be brought about by Personal Software Process initiatives (Humphrey, 1996). Though CMM pays attention to the role of "baselines", it remains a general framework. Here, we want to draw attention to specific issues which pertain to the use of architectures. Subsequently, the following measures are discussed:

- establish a mature planning
- standardise the development process
- create a design orientation

5.2 Mature Planning

Controlled system growth requires reliable planning of changes (Genuchten, 1991). Demarcation of responsibilities plays a crucial role here. Often it is only keyword-oriented, and not enough measures are taken to prevent duplications or omissions in the system. Interfaces are not worked out until a late stage. To prevent duplication during development, the functionality of packages and modules should be less broadly defined before the responsibility for their development is delegated to a particular group of developers. Functionality should be worked out with sufficient detail at a central level, through meetings of key developers and consultants.

The specification of the required functionality can result in a *business driven architecture*. The most important characteristic of this architecture should be that it sums up the "functionality units" of the complete system. A modular design approach could be followed to create this architecture. The architecture should be split up into pieces which can be assigned as objectives for a group of developers. What is going to be developed by whom and when can be established centrally this way. Obscurities about mutual prerogatives can only partly be solved through informal communication. A specification has to be available to formally outline the duties of each team of developers and describe the interfaces of their work (Berg and Wortmann, 1995; Pels et al, 1995).

5.3 Standardisation of the Development Process

Partly due to insufficient overall planning, the development process is currently too much feature-oriented (as opposed to design-oriented) and too unco-ordinated. Since delegation of development activities is driven by keywords and the functionality units are far from self-explanatory, developers occasionally receive guidance from consultants who tell them which features should be offered by that particular functionality unit. Beside its ad-hoc nature, this support is not problematic in itself, but without an overall design the development of the system as a whole is based on very local clues. Through the dialogue with (or under the pressure of) the consultant, the developer tends to include more and more features in his functionality unit, often regarding the official deadlines as unrealistic.

With this approach to development, it is very difficult to prevent that certain functionality is not covered at all or covered more than once, or that harmful links are made among tables and/or software. Even when it is known that a certain function is covered by a particular team, it is very tempting for other teams to still develop something similar when they need it for their own functions.

Matters would improve when teams of developers can be assigned to functionality units, based on the overall design and corresponding high-level specifications of the functionality (i.e. the business driven architecture). These teams should be supported by consultants, who provide the link with the market at a more detailed level. This procedure should be the same for each team of developers. Support from outside (customers, consultants) should be more uniform per development team. To improve the latter, the BaaN Company recently introduced their Customer Interaction Program.

5.4 Creation of A Design Orientation

To support many of the technical changes but certainly to sustain the validity of an architecture, management should establish a mind set among the developers, which is design-oriented. At the moment, this orientation has not pervaded most organisations sufficiently. One reason why many developers still think very much in "screens" may be that most customers and consultants look at the software this way. Not seldom, customers land a 700 page document on a desk, saying: "This is the output of our current system. Can your system give us the same output?"

To reduce the number of ad-hoc fixes, it should become a developer's second nature to reflect on a planned code change in terms of the architecture and its consequences for the system as a whole. In terms of the previous section, the developer should be made aware of his own domain. Likewise, he should be trained to contribute to the maintenance of the architecture. When significant code changes are not recorded as possible corresponding modifications of the architecture, the latter will be outdated and useless in no time.

6. Conclusion

On the basis of an action research project within the BaaN Company, we have discussed three aspects of enhancing controlled growth of information systems. Adequate technical decoupling streamlines the integration between system components, whereas conceptual decoupling can be used to secure integrity of the system. Over time, such exercises are only worthwhile when they are accompanied by initiatives to increase organisational maturity.

Acknowledgments

We are greatly indebted to Rob Nouwt (BaaN Development), and Martin Taal (BaaN Research) for their support in the Gordian project. In addition, our gratitude goes to Hans Wortmann for the initiation of the project, and Henk-Jan Pels for introducing us to the theory of modular decomposition. Finally, we would like to thank the staff at the BaaN Business IT School.

REFERENCES

VAN DEN BERG, R.J. and WORTMANN, J.C., **CIM Requires A New Manufacturing Engineering**, in P.P. Groumpos (Ed.) Proceedings of ASI'95, 1995.

VAN GENUCHTEN, M., **Towards A Software Factory**, Ph. D Thesis, Eindhoven University of Technology, 1991.

HUMPHREY, W.S. , **Managing the Software Process**, SEI Series in Software Engineering, ADDISON -WESLEY, 1990.

HUMPHREY, W.S., **Using A Defined and Measured Personal Software Process**, IEEE SOFTWARE, May 1996, pp. 77-88.

PELS, H.J., **Integrated Information Bases**, Ph.D Thesis Eindhoven University of Technology, 1988, (in Dutch).

PELS, H.J. and WORTMANN, J.C., **Modular Design of Integrated Databases in Production Management Systems**, JOURNAL OF PRODUCTION PLANNING AND CONTROL, Vol. 1, No. 3, 1990.

PELS, H.J., WORTMANN, J.C. and ZWEGERS, A. J. R., **Flexibility in Manufacturing: An Architectural Point of View**, in J.C. Wortmann (Ed.) Proceedings of the CIM at Work Conference , 1995.