Work in progress. Please do not cite nor circulate without the authors permission!

When Maintenance Becomes Use

Experiences from two industrial cases of EUD

Yvonne Dittrich [1,2], Lars Lundberg [1] and Olle Lindeberg [1]

[1] Blekinge Institute of Technology, Dept. of Software Engineering and Computer Science

[2] IT-University in Copenhagen, Department or Design and Use of IT

{yvonne.dittrich, lars.lundberg, olle.lindeberg}@bth.se

Abstract

To change applications to fit the needs for users, for different places and for development over time has long been a challenge called software maintenance. In this chapter we take up tailoring as a means to make software flexible. Starting with two case studies - one taking up tailoring for different users and one addressing changes over time - the article discusses the problems with both use and development of a tailorable application. To develop tailorable software raise new challenges; how to make a user friendly tailoring interface, how to decide what should be tailorable and how to make a software architecture that permits this, how to make the tailorable system having an acceptable performance. Our experiences also show that the boarders between maintenance and use become vague since tailorability can replace maintenance by professional software engineers with tailoring by advanced users. Based on the experiences from our two cases, we identify and discuss five important issues when designing and implementing tailorable systems in industrial settings.

1 Introduction

Tailorability – the 'light' version of End User Development allowing users to adjust and further develop a program during runtime – can be observed in many of today's applications. We all adjust the settings of our mail client, program the rules to sort mails into different folders or develop formats for our text processor. Though these applications have been around for quite a while there is little research addressing the software engineering of tailorable systems and the design in domains that require other non-functional qualities for the software besides flexibility and tailorability. This chapter reports results from two industrial cases of developing and maintaining tailorable software:

The Billing Gateway (BGw) sorts and distributes call data records produced by phone calls to billing, statistics and fraud detection systems. It provides an interface to tailor the sorting algorithms to the network the specific implementation of the BGw is part of and to changing business requirements, like new fraud indicators.

The contract handler is an in-house developed back-office system of a telecommunication provider, administrating contracts and computing payments based on certain events. The types of contract as well as the events that are subject of the contracts change permanently as the business develops. The contract handler case addresses 'design for change' rather than providing the possibility to adjust software to individual preferences.

Two features distinguish these two case studies from other research on tailorable systems. Research on commercial tailorable systems focuses so far mainly on the use and tailoring of such systems, and derives requirements and design implications from there or provides understanding of the social organization of, and around, tailoring activities. (see e.g. [14, 19]).

Work in progress. Please do not cite nor circulate without the authors permission!

Here our research provides additional results regarding the software development implications of designing for tailorability. We also consider the interaction between use, tailoring, maintenance and further development. Research addressing design issues is nearly exclusively using laboratory prototypes. (e.g. [12, 13, articles in this volume). When real use contexts are addressed, researchers often act as developers. (e.g. [15, 16, 17, 22, 23] and article in this volumes). Our findings partly confirm that the results of the existing research are also valid for commercially developed systems but the cases add to the design issues the interaction between flexibility to allow for tailoring and other software qualities like performance and reliability.

The research the chapter is based on relates to two different scientific discourses. In the Billing Gateway case, the research focused on performance issues and the deployment of parallel computing to improve it. Mathematical abstraction, algorithms, technical solutions and the evaluation of the changes in the real time behavior of the system are the means of argumentation in relation to this research discourse. The research around the contract and payment system focused not only on technical solutions but also on the software development practice and the interaction between users and developers necessary to achieve a fitting solution. For the research design a specific version of action research was applied [5]. The research methods respectively the involvement of the researchers will be detailed in the respective sections.

In this chapter we provide some answers to the question "What are the most important issues when designing and implementing tailorable software in industrial settings". In Section 3 we identify and discuss five important such issues. These issues are then summarized in Section 4. Section 2 describes our two industrial cases that serve as the basis for our conclusions.

2 Experiences

This section reports experiences from two different projects. Each of them is related to a sharp industrial application. In each of the projects we experimented with different solutions. Requirements from work and business contexts as well as from the technical context of the applications guided the evaluation of the respective prototypical solutions. In each case a specific solution optimizes the deployment of available technology according to the situated requirements and constraints. These solutions raise a set of issues that will be discussed in the following section.

2.1 Flexibility in Large Telecommunication Systems

In telecommunication networks different kinds of devices producing and processing data of different formats have to be integrated. For example, information about calls in a network is handled in call data records (CDRs). The CDRs are generated by Network Elements, such as telecom switches, and can contain information about call duration, originating telephone number, terminating telephone number, etc. The CDRs are sent to post processing systems (PPSs) in order to perform billing and fraud detection and so on. Both the kind and format of data and the way it is distributed to the different post-processing systems can vary and they change as the network develops. An additional difficulty in telecommunication networks is that the computation has to fulfill demanding real time constraints and that massive amount of data has to be handled.

The Ericsson Billing Gateway (BGw) is an example for such a performance demanding real-time telecommunication system but it also has to be customizable even after delivery. It

Work in progress. Please do not cite nor circulate without the authors permission!

functions as a mediation device connecting network elements with post processing systems like billing systems, statistical analysis and fraud detection.

Figure 1 shows the execution structure of the BGw. The structure consists of three major components: Data collection, Data processing, and Data distribution. Each component is implemented as one (multithreaded) Unix process. All data processing including filtering and formatting is done by the Processing process. The BGw can handle several different protocols and data formats. It can perform filtering, formatting and other processing of the call data.

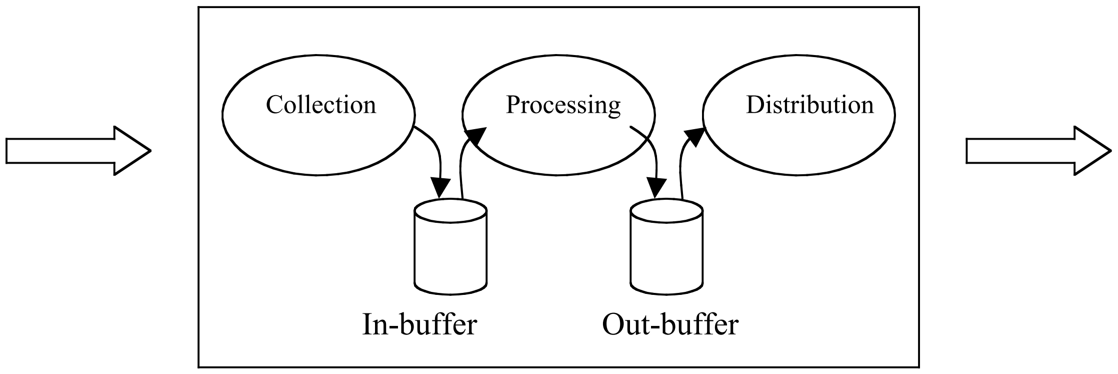

Figure 1 The execution structure of the BGw.

The data flow in the BGw is tailored using a graphical user interface. Icons are used to represent external systems. In Figure 2 files are retrieved from four Mobile Switching Centers (MSCs) of two different versions. CDRs from the Paging System are formatted to conform to the older version (“ver 7 -> ver 6” n Figure 2), and all CDRs are checked to see if the calls are billable. Billable CDRs are sent to the billing system and others are saved for statistical purposes. CDRs from roaming calls are also separated from other CDRs. For more information about the BGw architecture see [11].

Complementing the flexible definition of the dataflow, the BGw contains an interface that allows the tailoring of the filters and formatters that sort and re-format the incoming call data records to the interfaces of the post processing systems.

2.1.1 Methodology

The main methodology used is experimentation and performance evaluations of different versions of the software. These performance evaluations were done by Ericsson using real-world configurations. We have also conducted interviews with the designers of and the end users of the software. The software is used in about 100 places all over the world, and we have mainly had contact with users in Sweden, UK and Italy. The software developers are located in Ronneby in the south of Sweden.

2.1.2 Tailoring Filters and Formatters

All data processed by the BGw must be defined in ASN.1, a standard for defining data types in telecommunication applications. The BGw builds up internal object structures of the data called Data Units. One of the Billing Gateway’s strengths lies in the fact that the user can use a tailoring language – the ‘Data Unit Processing’ (DUP) language – to operate on those internal structures.

The sorting and reformatting is tailored through the definition of filter-, formatter, matching and rating nodes in the user interface. Such a node or component is represented in the user

Work in progress. Please do not cite nor circulate without the authors permission!

interface by an icon. Each node contains a script in the DUP language that is executed for every CDR passing through the BGw.

A filter node is used to filter out CDRs (e.g. IsBillable? in Figure 2). A filter node can, for example, filter out all roaming calls (a call made in a net other than the home net, e.g. when traveling in another country). The typical filter is rather simple and contains no more than around 10 lines of DUP code.

Sometimes it is necessary to convert a CDR from one format to another before sending it on to post processing systems (e.g. ver 7 -> ver 6 in Figure 2). The size, in lines of code, differs very much from one formatter to another. In its simplest form it might only change a couple of fields, whereas large formatters can contain several thousand lines of code.

CDR matching makes it possible to combine a number of CDRs into one CDR. It is possible to collect data produced in different network elements or at different points in time and combine them into one CDR. Matching nodes usually contain a lot of code, from a couple of hundred lines up to several thousand lines of code.

Rating makes it possible to put price tags or tariffs on CDRs. It can be divided into charging analysis and price setting. The main purpose of charging analysis is to define which tariff class that should be used for a CDR. The tariff class can depend on subscriber type, traffic activity, and so on. The price is then calculated based on the given tariff class, call duration and time for start of charge.

The DUP language is a mixture of C and ASN.1. It is a functional language that borrows its structure from C, while the way of using variables comes from ASN.1. An example of DUP code can be seen below:

```
CONST INTEGER add(a CONST INTEGER)
{ declare result INTEGER;
         result ::= 10;
         result += a;
         return result;}
```

Local variables can be declared at the beginning of a scope. A scope is enclosed by a '`{`' and a '`}`'. A variable is declared with the keyword `declare`.

declare <variable name> <variable type>

The drag-and-drop interface for defining the dataflow and the DUP language to define the processing of data in the different nodes gives the BGw flexibility to suit a wide range of applications without the need for re-compilation, since the functionality can be changed on-site by the customer.

Shortly after development performance became an issue for the BGw. Slow processing of CDRs was one problem in the BGw. Our research on the performance of the BGw has identified sever multiprocessor scaling problems [11]. Being a multi-threaded application, one expects that performance would improve when adding CPUs. This was, however, not the case.

Work in progress. Please do not cite nor circulate without the authors permission!

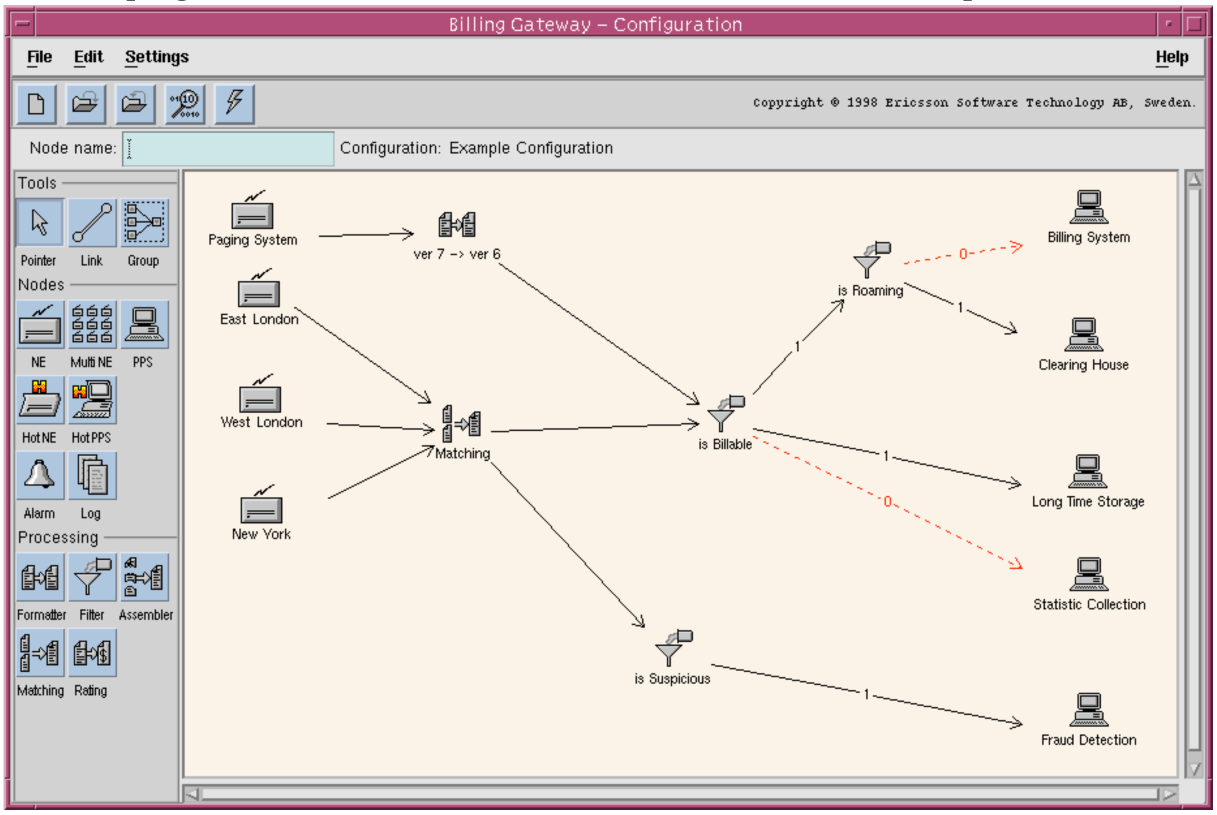

Figure 2: The Billing Gateway configuration view

2.1.3 Performance problems with the BGw.

The implementation of the DUP-language was identified as the major performance bottleneck and the main reason for the poor scaling [11]. The DUP language was implemented as an interpreted language which implies that each interpretation results in a serious of fuction calls, and it heavily uses dynamic memory, and thereby the shared heap. This again results in threads being locked on mutexes as multiple threads tries to allocate/deallocate dynamic memory simultaneously. By replacing interpretation with compilation (see next sub section), we were able to increase the performance with a factor of two on a single-processor. The compiled version also scales better, and that version was four times faster than the interpreted version when using a multiprocessor with eight processors.

The entire DUP implementation is accessed through three classes: DUPBuilder, DUPRouter, and DUPFormatter. A fourth class called DUProcessing is used by these classes.

The DUPBuilder uses an autmatically generated parser to build a syntax tree of the DUP-script. This means that a tree structure of C++ objects is created that represents the DUP source code. The DUPBuilder returns a DUProcessing object, which is the root node in the syntax tree.

2.1.4 Using Compilation instead of Interpretation

The interpretation of the DUP-scripts introduced a lot of overhead that degraded the performance of BGw. The attempts to solve this problem by using parallel execution were only partly successful. In order to get to the root of the problem we wanted to remove the interpretation.

Work in progress. Please do not cite nor circulate without the authors permission!

The obvious alternative to interpretation is compilation. Building a complete compiler is a major task, so we decided to build a compiler that translates the DUP-scripts into C++ code and then use an ordinary C++ compiler to produce the binary code. The binary code is then included in the program by using dynamic linking.

It turned out that the compiled version improved the average performance of BGw with a factor of two when using a uni-processor. The compiled version also scaled much better than the version using the DUP interpreter, and the performance of compiled version was eight times better than the version using the interpreter on a multiprocessor with eight processors. In fact, the compiled version scales almost linearly with the number of processors.

2.1.5 What is development, what is maintenance, what is use?

The BGw itself consists of 100,000-200,000 lines of C++ code. A typical BGw configuration contains 5,000-50,000 lines of DUP code, and there are more than 100 BGw configurations in the world today. This means that the total amount of DUP code is much larger than the size of the BGw itself. The DUP scipts are sometimes written by people in the use organization and sometimes by people from the local Ericsson officies. The BGw itself is developed in Ronneby in the south of Sweden.

A lot of effort has been put into making the architecture of the BGw itself as maintainable as possible, i.e. the cost for future changes should be minimized. However, since most of the BGw related code is done close to the users and outside of the development organization in Sweden, the main cost for future changes will probably be related to changing and maintaining DUP-scripts and not C++ code. This means that the most important aspect for minimizing the cost for maintanance is to make it easy to change the DUP scripts. Since the DUP scripts is one of the main ways with which the user interacts with BGw, one can say that making it easier to change DUP scripts is almost the same as making the BGw more usable. This means that the the borders between maintainability and usabilty are very vague when a large part of the development effort is done close to or by the user.

2.2 *Design for Change*

Our second case is also situated in the telecommunication area, but concerns a different kind of software. We co-operated with a telecommunication provider – Vodafone Sweden – and a small software developing company around the development of a back office system. The program is not a standard application that has to be tailored to the needs of different customers – like the BGw. It is a special purpose business application developed by the in-house IT unit. The flexibility is introduced to be able to accommodate changes in the business area.

The application that is subject to the research co-operation is a system administrating certain payments. The system computes the payments based on contracts. They are triggered by events.[1] With an earlier application only payments based on a certain event could be handled automatically. The business practice requires payments based on other events as well as new contract types.

The old system used for computing the payments had been used for several years. It automated the then existing contract types and computed the respective payments. However it was not very maintainable and after a while the users had to administrate some new contract types by hand and compute the respective payments that way as well. When even more radical changes in the business where discussed, the redevelopment was decided.

[1] To protect the business interest of our industrial partner, we do not tell about the character of contracts.

Work in progress. Please do not cite nor circulate without the authors permission!

Even a new system accommodating the recent developments would soon be outdated when business developed further. Implementing a tailorable solution seemed a promising idea. With the help of prototypes we explored the possibility to program a flexible solution based on a meta-modeling database system developed by the other project partner. The program that now is used in the company combines tailoring features with high maintainability so that even changes in the business area that go beyond the tailoring possibilities can be accommodated in an acceptable time frame. Based on this solution we developed a new prototype using meta-object protocol to implement the tailoring features. The sections 2.2.2 and 2.2.3 present and discuss the latter two solutions. Section 2.2.1 presents our research approach. To be able to understand the designs a conceptual model of the contract handler is presented first.

The system can be regarded as two loosely connected parts (Figure 3): the transaction handler and the contract handler. The transaction-handler application handles the actual payments and also produces reports while its database stores data about the triggering events, payments and historical data about past payments. (1)[2] The data describing the triggering events is periodically imported from another system. (2) To compute the payments, the transaction handler calls a stored procedure in the contract handler's database. (3) The event is matched with the contracts; several hits may occur. Some of the contracts cancel others; some are paid out in parallel. We call the process of deciding which contracts to pay 'prioritization'. (4) The result is returned to the transaction handler. (5) Payment is made by sending a file to the economic system.

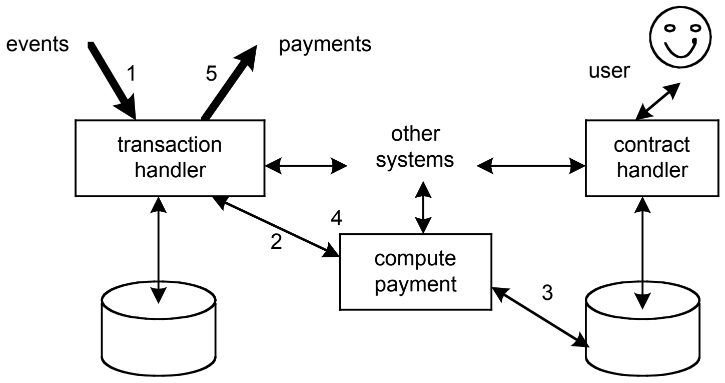

Figure 3

In order to make the system adaptable for future changes a conceptual model that facilitates a meta-model description of the system is needed. We first noted that a condition is meaningful in a contract only if the transaction handler can evaluate it when payment is due. This leads to the concept of event types; a payment is triggered by an event and all contract types belong to a particular event type. Each event type has a set of attributes associated with it that limit what a contract for such events can be based on. Contract types that a handled similarly are put together in one group. Secondly, we split up the computation of payments into two consecutive parts: first find all matching contracts and thereafter select which to pay (prioritization).

2.2.1 Research methods

To understand the kind of changes that might be required of the new contract handler, we reconstructed the history of the development and change of the software that should be replaced by the application under development. At the same time we started with participatory

[2] The numbers refer to Figure 3.

Work in progress. Please do not cite nor circulate without the authors permission!

observation of the development project. We took part and taped meetings during the prestudy. During the implementation of the software one of us did some minor programming task to be closer to the now more intensive technical design and implementation. We contributed to the project by exploring possibilities for a flexible design together with project members: participatory design workshops with the users took place to understand, how a tailoring interface for the new application could look like. We organised two workshops with the involved software developers to introduce meta-modelling techniques that are necessary one way or another when designing for flexibility. Together we implemented a set of prototypes proving that an extremely flexible implementation of the application was possible using the flexible database. [9] Based on the experience with the design a meta-object protocol version of the contract handler was developed at the university. As the development project at the telecommunication provider was about to be concluded, we never systematically evaluated that version together.

Our involvement into the project can be seen as a specific interpretation of action research, we tentatively call co-operative method development. [5] Empirical observation of practice, workshops and implementation of methodological innovation build a learning cycle that allows for a reflected change of software development practice and give the researchers feed back regarding the usefulness and applicability of methods.

2.2.2 Flexibility Light

The design of the finally implemented contract handler incorporates some meta-modeling features while using a normal relational database. The result was a flexible system without using any complex or nonstandard software. Flexibility here means both tailorability and maintainability, The flexibility comes primary from five features in the design.

The most important design decision was to modularize the contract-types internally as aggregations of objects handling single parameters and to group them according to the kinds of parameters that defined them. In most cases the program could handle all contracts belonging to the same contract type group in a uniform way, simplifying the program.

The second feature was to use the object oriented capabilities in PowerBuilder which was used to build the graphical user interface and the client side part of the application. Powerbuilder is a 4th-generation rapid development tool based on C++. The user interface is constructed with one window for each contract-group type. Different contract-group types have different set of parameters but each parameter type occurs in several contract-group types. Some of the parameters are in themselves rather complicated objects and the interfaces for them are also complicated. To simplify the system each interface for a parameter was constructed as an object, an interface object. The interface for each contract-type group was built as a set of interface objects. Since the parameters are treated in the same way in all contracts, this reduces the effort required to construct the interfaces and facilitates addition of new ones. The interface objects also guarantee that the user interface handles parameters in a consistent way.

The third is to use a non-normalized database. The contract types all have different parameters but they where nevertheless stored in the same database table which had fields for all parameters in any contract. This made a sparse table wasting some disc space, but allowed for a unified access. It would have been complicated to construct the interface objects otherwise.

As a fourth design feature, part of the computation is steered by a table indicating which contract type belongs to which contract type group. A new contract type that could be described as belonging to a contract type group would only require an additional line in that

Work in progress. Please do not cite nor circulate without the authors permission!

table. But even if a new contract type does not fit into one of the groups, it would only require minor programming effort as the interface object and the database are already there.

Last, but not least, the prioritization between different contracts triggered by the same event is controlled by a list describing which contract takes priority over the other. This way the earlier hard coded prioritization can be controlled more flexible.

The design combines different techniques to implementing for flexibility. When regarding the specific situation with respect to use, operation and maintenance of the system the overall evaluation was that design fitted well with the specific contexts of use and development at the telecommunication provider. Especially it was evaluated as better fitting to the situation at hand than a fictive fully flexible system utilizing the above-mentioned flexible data base system. To our surprise a comparison between the 'flexibility light' solution turned out to provide equal flexibility for anticipated changes though in some cases software engineers would have to do the adaptation. In cases where the system supported by the prototypes would have been better, the changes would require changes in other systems as well. Regarding usability, maintainability and reliability the 'flexibility light' solution was evaluated better. [9]

2.2.3 Using Meta Object Protocol to separate tailoring and normal use features in the software architecture

One of the reasons for the low maintainability and reliability of the more flexible solution was the interweaving of the meta-level that described the structure of the contracts and the base level to access the concrete data throughout the code. E.g. the metadata about the structure of the contract type steered the layout on the screen. To explore whether it is possible to develop a better designed system supporting maintenance, testing and debugging as well as flexibility we implemented a prototype inspired by Kiscales' meta-object protocol. [8] We also deployed the idea of designing the contracts as aggregations of building blocks, each modeling a specific parameter from the 'flexibility light' solution.

The metaobject protocol is based on the idea to open up the implementation of programming languages so that the developer is able to adjust the implementation to fit his or her needs. This idea has subsequently been generalized to systems other than compilers and programming language. [8] Any system that is constructed as a service to be used of client applications (as for example operation systems or database servers) can provide two interfaces; a base-level interface and a meta-level interface. The base-level interface gives access to the functionality of the underlying system and through the meta-level interface it is possible to alter special aspects of the underlying implementation of the system so that it suits the needs of the client application. The meta-level interface is called the metaobject protocol (MOP). The prototype uses this idea to separate tailoring and use in the software architecture and allow for a better structured design of both the tailoring features and the base functionality. [10]

For each value in the event data that can be used as a constraint in a contract, a class is defined that takes care of the functionality for the constraint; displaying it for viewing and editing, checking the consistency of the input data, storing and retrieving the specific values from the database. A contract is implemented as an aggregation of a number of such objects plus a set of objects handling mandatory contract specific data such as contract ID, creation date, validity dates, versioning information and so on. Each contract type is defined by a class that specifies what constraints an event has to satisfy to trigger contracts belonging to this type.

Work in progress. Please do not cite nor circulate without the authors permission!

New classes defining new contract types can be created in the meta-level interface. They are put together by selecting possible constraints from a menu. The meta-level also offers possibilities to change existing contract type classes and to define additional classes for implementing constraints. That way it is possible to adapt the new contract types to requirements that have not been anticipated yet. The meta level is implemented using a metaobject protocol: Through menu driven selection the user assembles a new contract type. The class describing this contract type is written as Java code to a file. After translation the class becomes part of the base level without even restarting the system.

The new contract type is available in the menu of the base level program. The base level program consists of a frame providing access to the contracts and to the existing contract types when designing a new contract. Besides some use of the Java reflection features the base-level program is a ordinary Java program. In case of errors or other problems it is easy to test and to debug. A system constructed according to this idea can be implemented with a traditional, sparsely populated database or with a database system that allows for changing the data model during runtime.

The meta-object protocol design is a mean to separate concerns. Business logic regarding the contracts and constraints is implemented in the building blocks and in the base level of the program. In some cases we may be dissatisfied with the resulting contract types, e.g. we may want a user interface that is specially constructed for just this contract type. With the metaobject protocol design this can easily be solved by using a hand-coded class instead of the automatically generated class for such a contract type - we are free to mix automatically generated classes and hand-coded classes. Also special business logic can be incorporated in the program this way. The business logic guiding and constraining the assembly of contract types can be implemented in the meta level part of the program.

The main advantaged with the metaobject protocol design is the separation of concerns. The base level of the program is just a normal program where some parts are automatically generated code. In the same way the meta-level of the program is only concerned with the tailoring functionality. The functionality of the meta-level part can be tested independently. As the base program works as a normal program, it can be tested and debugged as usual. One could even think about developing specific test cases together with creating a new contract type class. We used Java for implementing of the prototype. The meta-object protocol solution is possible based on standard software and a standard database system.

As the flexibility light solution was already taken in use when we finished the new prototype, we did not evaluate the meta-object prototype solution together with the telecom provider. However, we consider this solution as addressing some of the drawbacks of we identified during the evaluation of the early prototypes, and it would be more flexible than the ‘flexibility light’ that was implemented at the telecommunication provider.

2.2.4 Tailoring, software evolution and infrastructures.

The contract handler example shows that even relative simple business applications must be flexible when supporting a developing business domain. It also shows that whatever kinds of changes can be anticipated, to accomodate unanticipated developemnts, use and tailoring might have to be interleaved with maintenance and further development by software engineers.

The decision what to implement as tailoring functionality and what to leave to the maintenance by software engineers depended in our case on the well established co-operation between the IT-unit and the business units of the company. A similar design would not have

Work in progress. Please do not cite nor circulate without the authors permission!

been acceptable if the development would have been outsourced as the group manager of the IT unit responsible for the development remarked during a project meeting.

A third issue that became visible when evaluating the change efforts for the flexibility light version contra a fully flexible system is that business systems in such data-intensive business domains are often part of an infrastructure of networked applications. Changes in the business practice often imply changes in more than one program or in the interplay between different applications. Here 'design for change' implies the sustainable development of heterogeneous infrastructures.

3 Challenges, Problems and Solutions

The two cases above show that tailorable software is an issue not only as a means to tailor the user interface of a standard system, but also in industrial development and deployment of software. It allows to delay design decisions until after the program is taken into use and to adapt software to changing business and user requirements. The two very different cases provide quite a spectrum of experiences. In this section we summarize the lessons learned so far and the challenges for software design and development we observed.

3.1 Usability of the tailoring interface

As normal interfaces, tailoring interfaces have to be understandable from a users' perspective. They have to represent the computational possibilities not only in a way that makes them accessible for use, but also helps the user to understand how to combine them. That also implies that the tailorable aspects of the software have to be designed - even on the architecture level – to match a use perspective on the domain. The tailoring interface has to present the building blocks and the possible connections between them in a comprehensible way as well. Mørch's application units [12,13] and Stiemerling et al's component based approach [16] are examples for such architecture concepts. Stevens and Wulf discuss this issue when designing a component decomposition of a tailorable access control system [15]. This issue relates to a discussion regarding the relationship of the design of the software architecture and the structure of the user interface. E.g. Zuelleghoven et al. [r24] developed an approach to design of interactive systems that relates the architectural design and the user interface by using tools and materials a common design metaphor. However, designing the contract handler according to this approach would not automatically have led to an architectural design that supported tailorability.

In the Billing Gateway the data flow interface provides a very intuitive interface from the user's point of view. Also the language for tailoring filters and formatters relates well to the technical education of its users. Nonetheless, end-users have shown some reluctance to tailor the application, and the tailoring activities were in many cases done by software engineers. Also in the contract handler the limited tailoring capabilities were presented to the users in a form close to their own concepts. But here the users quit early declared that they did not want make changes in the system by themselves. They felt more comfortable by letting the software engineers responsible for system maintenance doing the tailoring. In the latter case the users felt insecure regarding the correctness of the results of the adaptation. The latter is further discussed in section 3.4.

The experiences from our two cases show that the challenge is to find ways to structure the tailoring capabilities so that it is easy to use and understand for the user, while at the same time providing tailoring capabilities that are powerful enough to provided the desired flexibility. In the two cases considered here, the end users have (at least initially) felt that the tailoring activities should be left to software engineers. However, we believe that the users'

Work in progress. Please do not cite nor circulate without the authors permission!

need for software engineering support will decrease as they get more used to the systems. Experiences from widespread and less powerful tailoring, such as adjusting the settings in mail programs and providing formulas in spreadsheet applications, show that the support of software engineers is clearly not needed in those cases. We believe that for applications that are used by a large number of people with very different backgrounds, e.g. mail and spreadsheet programs, the trade-off between ease of use and powerful tailoring capabilities, must be different from what we see in the two more specialized applications studied here. The systems considered here will be used by a relatively small number of people and it is thus reasonable to give higher priority to powerful tailoring, even if the users initially require specialized training and/or support from software engineers.

One result of introducing powerful tailoring interfaces for certain specialized applications is that the user will become somewhat of a domain specific developer. It also means that new versions of the applications will not always be done by reengineering the software itself; it will, to an increasing extent, be done by advanced tailoring. What was previous software development and maintenance will thus become tailoring and use.

3.2 Deciding what should be adaptable and how to design for it.

The requirements a software program should fulfill are hard to determine in advance, particularly adaptable software. In both our cases, the development organization had built similar software before. Ericsson has a long history in telecommunication; the decision to make the Billing Gateway tailorable was an answer to an increasing effort of building different versions of the system to different customers. The contract handler was a re-development. The experience with the changing requirements was the main motivation to look into the design of adaptable software.

Part of this design problem is the difficulty to anticipate changes to provide for. [6, 16, 18] Domain knowledge and feedback from use is important to understand where flexibility is needed.

However, when deciding what is fixed and how the adaptable parts look like, one implicitly decides the architecture of the whole system. In the Billing Gateway case, designing filters and formatters as programmable manipulation of the dataflow also defined the basic structure of the system and vice versa. For the contract handler design, the identification of fixed building blocks made it possible to implement the contracts as assemblies of these blocks. One can also turn this reasoning the other way; it is first when a basic conceptual model of how the system is developed that we can understand what can be made tailorable. An example of this is in the contract handler, only when understanding contracts as sets of constraints to be matched by events, one can define the constraints as building blocks to put together a contract. In both cases the design of the stable and the adaptable aspects were dependent on each other. As in architectural design, the design of the parts is only understandable in relation to the whole.

The evaluation of the flexibility light solution made visible that also the organization of the software development influences the design decision. [4] In-house development allows for leaving part of regular adaptation to the software engineers. Outsourced development would have led to other design decisions. Users of off-the-shelf software cannot rely on such co-operation with the developers of the software either. Here users depend on a more comprehensive tailoring interface for the necessary adaptations.

The challenge here is thus to find a good balance between what future adaptations of a certain software should be made tailorable for the end user and what future adaptations should be left to software engineers that redesign and maintain the software itself. Leaving everything (or at

Work in progress. Please do not cite nor circulate without the authors permission!

least very much) to the end user will cause problems since this will require very advanced tailoring (which in turn may require help from software engineers) and which in turn might make testing and documentation difficult, and thus generate problems with the reliability of the system. On the other hand, leaving very much to the software engineer will significantly increase the time and cost for introducing new functionality. We believe that the trend is that more and more functionality is made tailorable; the two systems studied here are examples of that. However, it is important to find a good balance between traditional software maintenance, done by software engineers, and tailoring done by the users.

There is no systematization of design for tailoring and EUD yet. Here perhaps a collection of high-level design patterns might slowly lead to a more systematic overview of different possibilities and their strengths and weaknesses. Our cases indicate that the evaluation of solutions, that have a good balance between tailoring and traditional software maintenance, have to take the way use and development relate into account as well as the interplay of tailorability and non-functional requirements.

3.3 Performance

Many design techniques to provide flexibility and tailorability of software decrease the performance of the program. Especially in the Billing Gateway case this has been a problem. However, this problem often can be taken care of by using good technical solution. In the BGw we first tried to improve performance by using multiprocessor technology. This approach only resulted in limited performance improvements. It turned out that the best way to handle the problem was to replace interpretation with compilation combined with dynamic linking, thus maintaining the flexibility and tailorability and improving the performance.

Experiences from a very flexible and tailorable database system showed that performance was initially a factor of 10-20 lower than a traditional system. The reason for this was that the system used one level of indirection, i.e. the system first had to look in a data base table for meta data before it could interpret any actual data values. This performance problem was removed by introducing some controlled redundancy which decreased the slow down from a factor 10-20 to a factor of two [2].

These experiences show that the performance problems due to flexibility and tailorability are often possible to handle without too much trouble. Flexible and tailorable software can thus be used also in performance demanding real-time applications like the Billing Gateway system.

3.4 Software engineering education and tools for end-users?

The Billing Gateway was designed to be adapted by the telecommunication providers themselves. Being technical educated people and well acquainted with the standards used for the description of the call data records, the DUP-language should not provide a major problem. However, the adaptations were mainly performed by Ericsson personal. The reason was that the end-users were afraid to implement erroneous tailoring constructs and cause major loss of money for their organization. The users of the contract handler had similar reasons for refusing to tailor the program. They did not want to be responsible for causing loss of money and reputation for the telecommunication provider by making mistakes when doing tailoring. The developers had access to test environments and tools used for ordinary software development and were able to check whether the change had the intended effect. Also in the contract handler case, the users were reluctant to take responsibility for the tailoring. An interesting question is if better tools and education in testing and documenting could be of help for the users. From other researchers similar needs for documenting and testing of

Work in progress. Please do not cite nor circulate without the authors permission!

tailoring constructs have been reported. [7, 16] Wulf [22, 23] proposes exploration environments that allow for save trial and error. Burnett [1] explores constraints based testing and verification support for end-users. In the Meta-object version of the contract handler the definition of contract types that do not make sense from a business point of view can be prohibited.

In our cases, even analysis and design came up as an issue for tailoring. 'If the system can handle any kind of contract, how do we decide on which kind of contracts we want to offer?' a manager from the business unit asked during a workshop when confronted with a mock-up showing a possible interface of a tailorable system. Involving and paying the IT unit that is run as an internal profit center provided enough effort to deliberate new contract types from a business point of view. Trigg and Bødker [19] observed the development of an organizational infrastructure around the deployment of the tailoring features of a text editor in a government agency. The design of form letters used instead of individual formulations provoked the establishment of a committee that included even legal experts to decide on and review the form letters to be taken into use.

Other authors have observed the need to document the tailoring constructs, to keep different versions and to share them [6, 16]. Such features will be necessary even when tailoring the common artifact – like in the contract handler case – or the infrastructure of a business – like the Billing Gateway allows. It seems if the user is provided with the means to tailor the software, she also has to be provided with the means and tools to deliberate, to document, to test and to reuse the tailoring constructs. Especially testing and documentation has to be provided for tailorability and EUD to be deployed more systematically in commercial contexts. Only experience will tell about the extent and kind of such 'end-user software engineering' that is needed. We do, however, believe that users of tailorable systems will to a growing extent need better 'software engineering' education and especially tools since they might become a kind of domain specific software developers.

3.5 How does software development change in connection with tailoring?

Developing software that is adaptable for different ways of using it or for a developing business area changes software engineering . To design a solution for a problem is no longer enough. One has to design spaces for adaptation to a set of diverse uses and anticipatable changes. Additionally, one has to consciously defer part of the design to the user.

Tailorability allows the users to implement adaptations that otherwise would be subject to maintenance. For the Billing Gateway, the tailoring constructs can become a major part of the overall code developed. On the one hand, maintenance effort is traded for tailoring effort for the users. Already [6] and [21] mention that tailorability rationalizes development as it prolongs the maintenance cycle. On the other hand, the design of a tailorable system may be more complex, especially when it faces performance issues. Making a program tailorable will thus also shifts the software engineering effort from the maintenance phase into the design phase and not only from maintenance by professional software engineers into tailoring by advanced users.

Even if we can hope for less maintenance when designing system tailorable there will always be need for maintenance when change requirements come up that cannot be accommodated by the adaptability the design provides. Tailorability then will raise a new set of problem. A new version of the system will have to allow the users to keep the adaptations they have done

Work in progress. Please do not cite nor circulate without the authors permission!

in the old system. With the installation of a new version, not only the old data has to be translated, but also the tailoring constructs. This area is in the need of more research.

The life cycle of software will have to accommodate for the interlace of development, use, tailoring, small maintenance tasks and major re-development. This might to imply more flexible ways of organizing software development, and a less rigid divide between use and development of software. This will require increased communication between the software engineers and the advanced users. Already 1991 Trigg anticipated that development. [18] Our research indicates that this is the case when developing software that is part of an infrastructure for developing work and business practices. [3]

4 Conclusions

Henderson & Kyng [6] in their article "There's no place like home: Continuing design in use" takes up three reasons for doing tailoring; "the situation of use changes", "it [is] difficult to anticipate [the use]" and when "creating a product which will be purchased by many people". In this chapter we have exemplified the first and last of these and we believe that tailoring has an essential role to play in industrial software development to solve these problems. When it comes to the problem of anticipating how the system will be used tailoring is certainly a possibility to alleviate this problem.

Based on our experience from the two cases discussed here we have identified a number of important issues when designing and implementing tailorable systems in industrial settings:

- The balance between providing a tailoring interface which is easy to use while still providing powerful tailoring possibilities. Our conclusion is that it makes sense to give higher priority to powerful tailoring in specialized applications, like the ones studied here, compared to more general applications, e.g. mail programs.
- The balance between traditional software (re-)development and maintenance on the one hand and tailoring and use on the other. Our conclusion here is that the trend is towards handling more and more of the need for future adaptability by tailoring.
- The "conflict" between tailorability and flexibility on the one hand and performance on the other. Our conclusion here is that this problem can, in most cases, be solved by innovative technical solutions, and tailoring can thus be used also in performance demanding real time applications.
- There is a need for giving the end users better tools for testing, documentation and reuse/sharing of tailoring constructs as well as the necessary education to use them. This is particularly true for users of specialized tailorable applications, like the ones studied here. Such users might in fact become some kind of domain specific software developers.
- Software maintenance and (re-)development will to a growing extent be mixed and interlaced with tailoring. This will require increased communication between the software engineers and the advanced users.

Acknowledgements

References

1. Burnett, M. This volume.
2. Diestelkamp W. and Lundberg L., Performance evaluation of a generic database system, International Journal of Computers and Their Applications, Vol. 7, No. 3, Sep. 2000, pp. 122-129.

Work in progress. Please do not cite nor circulate without the authors permission!

3. Dittrich, Y. and Lindeberg, O. How Use-Oriented Development can take Place. Information and Software Technology, Vol. 46, Issue 9, July 2004, pp. 603-617.
4. Dittrich, Y. and Lindeberg, O. Designing for Changing Work and Business Practices. In Patel, N. (ed.) Evolutionary and Adaptive Information Systems. IDEA group publishing, 2002
5. Dittrich, Y,. Doing empirical research in Software Engineering – Finding a path between understanding, intervention and method development, In Y. Dittrich, C. Floyd, R. Klischewski (Eds.), Social Thinking – Software Practice, The MIT Press, Cambridge, USA, 2002.
6. Henderson, A., & Kyng, M., There is no place like Home: Continuing Design in Use. In J. Greenbaum, & M. Kyng (Eds.), Design at Work. Lawrence Erlbaum Associates, 1991, pp. 219-240.
7. Kahler, Helge (2001): Supporting Collaborative Tailoring. Ph.D. thesis. Roskilde University. Datalogiske Skrifter, ISSN 0109-9779 No. 91, 232 pages.
8. Kiczales, Gregor 1992: "Towards a New Model of Abstraction in the Engineering of Software", in Proceedings of International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture, Tama City, Tokyo, November 1992.
9. Lindeberg, O. and Diestelkamp W. How Much Adaptability do You need? Evaluating Meta-modeling Techniques for Adaptable Special Purpose Systems. In *Proceedings of the Fifth IASTED International Conference on Software Engineering and Applications*, SEA 2001.
10. Lindeberg, Olle & Eriksson Jeanette & Dittrich, Yvonne 2002: "Using Metaobject Protocol to Implement Tailoring; Possibilities and Problems", in The 6th World Conference on Integrated Design & Process Technology (IDPT '02), Pasadena, USA, 2002.
11. Mejstad, V., Tångby, K.-J. and Lundberg, L. Improving Multiprocessor Performance of a Large Telecommunication System by Replacing Interpretation with Compilation. In Proceedings of the 9th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, April 2002, Lund Sweden, pp. 77-85.
12. Mørch, Anders I. 2003:" Tailoring as Collaboration: The Mediating Role of Multiple Representations and Application Units", in N. Patel: Adaptive Evolutionary Information Systems. Idea group Inc. 2003.
13. Mørch, Anders I. & Mehandjiev, Nikolay D. 2000:" Tailoring as Collaboration: The Mediating Role of Multiple Representations and Application Units", in Computer Supported Work 9:75-100, Kluwer Academic Publishers.
14. Nardi, B.A. A Small Matter of Programming. MIT Press 1993.
15. Stevens, G., Wulf, V. A New Dimension in Access Control: Studying Maintenance Engineering across Organisational Boundaries. Proceedings of the CSCW 02, November 16–20, 2002, New Orleans, Louisiana, USA.
16. Stiemerling, Oliver, Kahler, H. & Wulf, V. 1997 How to make software softer- Designing tailorable applications. Proceedings of the Disigning Interactive Systems (DIS) 1997.
17. Stiemerling, Oliver, & Cremers, Armin B. 1998: "Tailorable component architectures for CSCW-systems" in Parallel and Distributed Processing, 1998. PDP '98. Proceedings of the Sixth Euromicro Workshop pp: 302-308, IEEE Comput. Soc.
18. Trigg, R. Participatory Design meets the MOP: Accountability in the design of tailorable computer Systems. In: Bjerkness, G. Bratteteig, G., and Kauts, K. (eds.) Proceedings of the 15th IRIS, Department of Informatics, University of Oslo, August 1992.

Work in progress. Please do not cite nor circulate without the authors permission!

19. Trigg, R., & Bødker, S. From Implementation to Design: Tailoring and the Emergence of Systematization in CSCW. Proceedings of the CSCW '94, ACM-Press, New York, 1994, pp. 45-55.
20. Truex, D. P., Baskerville, R., & Klein, H. 1999 Growing Systems in Emergent Organisations. Communications of the ACM vol. 42, pp. 117-123.
21. Wulf, V., Rohde, M. Towards an Integrated Organization and Technology Development. In: Proceedings of the Symposium on Designing Interactive Systems, 23.8.-25.8.1995, ann Arbor (Michigan), ACM Press, New Yorck 1995, S. 55-64.
22. Wulf, V. " Let's see your sSearch Tool!" On the Collaborative use of Tailored Artifacts. Proceedings of the GROUP '99 conference, ACM Press, New Yorck, pp 50-60.
23. Wulf, V. Exploration Environments: Supporting Users to Learn Groupware Functions. Interacting with Computers, Vol. 13, No.2, 2000, pp. 265-299.
24. Zuellighoven H. Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz; dpunkt-Verlag Heidelberg, 1998.