Separation of Concerns: A Case Study

Jilles van Gurp & Jan Bosch

University of Groningen
Dept. of Mathematics and Computing Science
PO Box 800, 9700 AV Groningen
The Netherlands
[jilles|bosch]@cs.rug.nl
http://segroup.cs.rug.nl/

Abstract. In this paper we present the results of a case study we conducted at two local SMEs (Small and Medium sized Enterprises) in two different domains. In the case study we examine how these companies handle separation of concerns in their respective domains. We focused on which concerns were perceived as problematic; what kind of design solutions were used to work around these problems and what the effect of these design solutions was on the separation of concerns. In our analysis we reflect on if, and how the use of advanced separation of concerns technology, such as e.g. Aspect Oriented Programming (AOP), would be useful. An important conclusion of our paper is that in both cases the companies do not need such tools for separating the concerns they are aware of (i.e. anticipated concerns) since adequate, conventional techniques have been applied. The benefit of applying such techniques to separate the remaining, unanticipated concerns is unclear as well since it is anticipated that at least some refactoring of the original system would be required in order to apply such techniques.

1 Introduction

To deal with the ever-increasing complexity a better separation of concerns in software systems is needed. In the separation of concerns community a number of problems regarding separation of concerns in large software systems are addressed. To deal with these problems such approaches as Aspect Oriented Programming [12] and Multi Dimensional Separation of Concerns [23] are currently being developed. These approaches give software developers the possibility to modularize their systems in a more suitable way.

1.1 Goal of this paper

The goal of this paper is to look at separation of concerns in the current practice of software engineering. Specifically, we are interested in how conventional design solutions are used to achieve separation of concerns, what the problems are and how they are addressed. A second topic in our study is if and how the use of advanced separation of concerns (ASOC) techniques could improve the separation of concerns.

1.2 Cases

We have conducted structured interviews at two SMEs (Small and Medium sized Enterprises) from two domains in software engineering; i.e. embedded systems and enterprise systems.

The first company where we conducted interviews, Rohill Engineering B.V. [20], is a worldwide operating and recognized company, specialized in product and system development for professional mobile communication infrastructure. Their products are typically used in communication networks for e.g. police departments, fire departments or taxi centrals. The hardware Rohill sells consists of off the shelf components. Their embedded software, however, is proprietary.

The second company is Vertis B.V. [24]. Vertis is an IT service provider that, among others, implements standard solutions such as Oracle based database products. The implementation process, depending on the needs of the customer, can be very extensive and typically also includes system installation, system administrative services and even training the staff to use the systems. Often systems are sold in combination with a service contract specifying e.g. how long the systems are to be maintained by Vertis.

By selecting companies from two domains, we intended to identify whether there are differences in the way concerns are dealt with in these domains as well as whether there are commonalities in the way certain concerns are dealt with.

1.3 Remainder

In the next section, we discuss the methodology we used for the case study in more detail. In Section 3 and Section 4, the two companies are discussed in detail. In Section 5, we reflect on the results and make some more general observations that are motivated using examples from the cases. In Section 6, we reflect on the validity of our case study and highlight its strengths and its weaknesses. Subsequently, we discuss related work in Section 7, and, finally, in Section 8 we conclude the paper.

2 Methodology

We have considered various ways of conducting our research. Considering the amount of time and resources available as well as the amount of effort that both companies were willing to put in our research we have chosen to do a number of structured interviews with key persons within the companies. Our approach consists of three steps:

- Preparation. At both companies the structured interviews were preceded by introductory meetings during which we presented ourselves, explained the research topic, our methodology and what we expected from them.
- Interviews. After the introductory meetings, appointments were made for conducting the structured interviews. An important advantage of structured interviews is that it requires a minimal amount of time from the interviewees. This is important since all of the persons we interviewed had busy jobs. In addition, a structured interview is less rigid than e.g. a questionnaire, which allows us to

retrieve company/domain specific information. In many cases, the questions were used as a starting point for discussion. Yet, because of the structure, we can compare the interviews. Finally, because we had two interviews at each company, we were able to validate the statements of the interviewees by comparing their responses.

` Feedback. The results of the interviews were returned to the interviewees who then had the opportunity to give feedback and correct things they did not agree with.

2.1 Terminology

As a part of the introductory meetings, we also discussed the notion of separation of concerns. Unfortunately, the people we interviewed generally were not aware of the academic terminology we commonly use to discuss this topic such as cross cutting features, aspect oriented programming, concerns, etc. In order to bypass this issue, we avoided using such terminology during the interviews and instead tried to formulate our questions in such a way that our interviewees could understand them and identify with the topic. Where applicable, we will provide explanations of terminology in the remainder of the paper.

2.2 Selection of concerns

The purpose of ASOC techniques such as Aspect Oriented Programming or Subject Oriented Programming [12][10] is to enable programmers to address concerns such as e.g. synchronization or logging separately and automate the integration of the separately developed concerns.
Since interview time and the knowledge of our interviewees about ASOC were limited, we selected a handful of well-understood concerns that typically are of consideration to some extent in software development. Rather than asking technical questions about these concerns (which would require lengthy explanation of the various terminology and influence the interviewees), we instead selected a few associated quality attributes. This allowed us to bridge the gap in knowledge since the interviewees were reasonably knowledgeable about these quality attributes since they had to meet requirements related to these quality attributes. In each of the four interviews, we confronted the interviewees with questions about the five quality attributes discussed below.

2.2.1 Performance

Performance requirements can be specified and enforced in many ways. In a real-time system it is quite common to specify performance requirements explicitly and to include functionality that monitors whether the requirements are met. In other systems, the requirements may be somewhat more flexible. Especially real-time related functionality (e.g. enforcing constraints) is generally seen as crosscutting [1][12]. In the interviews we tried to recover in what way performance requirements influenced implementations and how performance requirements were tested. We asked the interviewees to identify a few typical performance requirements. Then we asked how they assessed whether these requirements were met and if so, how assessments were made with regard to performance. In the Rohill case we also studied performance related design decisions.

2.2.2 Maintainability

A bad separation of concerns translates into higher maintenance cost because when changes affect such concerns, they typically affect large parts of the source code. Consequently we included maintainability as a quality attribute in the case study. We asked the interviewees about the size of a typical system. We also asked about maintenance effort and if there were any bad practices that were actively fought against (e.g. multiple inheritance). Finally we asked whether there was a review process and whether they were familiar with/used refactoring techniques such as those discussed in [6].

2.2.3 Concurrency/synchronization

Concurrency related functionality, in particular synchronization code, is often used as an example case for crosscutting features. For example, Kiczalez et al. use synchronization constraints as an example of aspects [12]. Consequently, we included the synchronization concern in our study to verify whether this was a real concern in our cases. We asked what the impact of synchronization issues such as synchronization code was on the software system. What percentage of the code was affected by it and whether this code was a source of bugs.

2.2.4 Flexibility

A flexible system makes it easy to make certain kind of changes to it. Flexibility is not the same as maintainability and often there are conflicting requirements with respect to both quality requirements. For instance, flexibility-enhancing mechanisms (e.g. abstract classes) make a system more complex and consequently maintainability is affected negatively. By making a system flexible, the concerns that are typically affected by changes are separated in such a way that such changes are easy. In order to find out whether our interviewees were successful in separating concerns, we asked them whether they used design patterns to make code flexible. Further more we asked whether flexibility was considered during design.

2.2.5 Reliability

Low reliability is another symptom of bad separation of concerns. Memory management is a good example of a concern that is typically problematic with respect to reliability. Since we suspected in both cases reliability would be important, we asked how reliability requirements were specified and inquired about the translation of these requirements into code.

2.3 Interview structure

Since the domains of both cases are very different, the amount of time we spent discussing these concerns varied between the interviews. For each concern we prepared a set of questions that were used in each interview. However, when necessary, we asked additional questions or skipped questions that were not applicable.

3 Case 1: Rohill

TetraNode is a multi protocol backbone for mobile communication networks that implements the Tetra standard. Tetra (TErrestrial Trunked RAdio) is an ETSI (the European Telecommunications Standards Institute [4]) standard for digital trunked mobile radio. Rohill is currently in an advanced stage of building this software product.

3.1 Application Domain

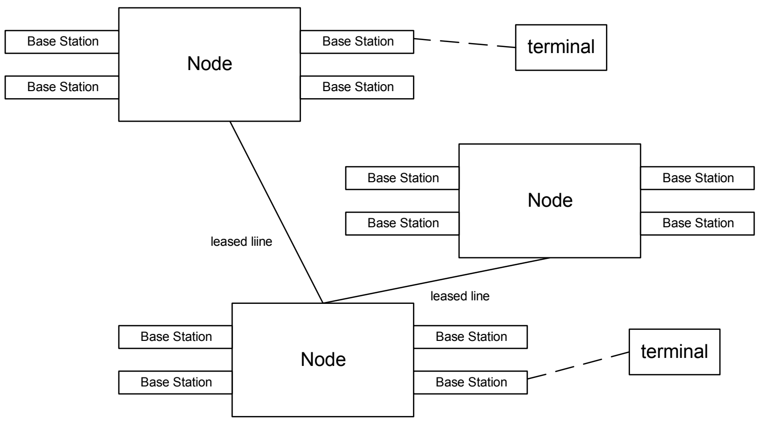

Fig. 1. A network of TetraNodes

A TetraNode network consists of nodes (usually a PC), base stations (connected to a node) and terminals (e.g. portable radio's used in ambulances), which are organized in a flat network topology not unlike the internet. When the implementation is finished, Rohill will sell the TetraNodes in various configurations, the terminal-devices are sold by third parties. The main advantage of the TetraNode network topology is scalability. Simply by adding nodes, the network can be extended. Each node consists of a standard PC and a subsystem with interfaces for communicating with base stations, over e.g. leased lines and interconnection with other nodes. Due to the flat network topology, a TetraNode network can scale to up to a million users.

At a first glance, TetraNode is very similar to the GSM network used for mobile phones (GSM is also an ETSI standard). However, there are significant differences in quality requirements. While both provide digital radio services, there are differences in network topology, amount of users, typical usage of the network, length of communication, and so on. As compared to GSM, the Tetra protocol has shorter connection times (less than 300ms), larger cells (the area around a base station) with fewer users that typically use the network for short calls to mostly other users in the same cell (i.e. the communication does not need inter-TetraNode connectivity for such calls). In addition security is also of importance when a TetraNode network is used by e.g. the police.

While the Rohill hardware platform consists of mostly of-the-shelf components, their software is proprietary and is indeed their competitive edge. The software consists of the TetraNode Foundation Classes (TFC), a support library developed internally; the implementation of the various protocol stacks and various applications on top of the protocols implementing such things as routing, database management, network management, etc. However, in this paper we will focus on the TFC and the protocol implementations since those components are in a relatively advanced stage of development.

3.2 Development Method

The development process at Rohill is iterative. Typically there is a new release every few months that is used for internal testing. Typically these tests involve loading the software onto test equipment and stress testing it with e.g. simulated attempts to set up calls.

3.3 Quality attributes

3.3.1 Performance.

The TetraNode hardware architecture consists of both embedded systems and standard PCs. A performance factor that needs to be taken into account is how much kilobits can be processed by the hardware when handling speech. Specialized FPGA chips are often used to improve throughput. On the PCs however, regular chips like e.g. Intel’s Pentium have been used as well. To maximize throughput, the software architecture is set up in such a way that data flows through the system efficiently. There are two notable design decisions that affect performance in the software:

The binary data packets that are inserted in a connection are assembled at the latest possible moment. This generative approach assures that this computationally expensive operation is executed only once and that there is no duplication of the data internally.

It was decided to use C++ templates to make the source code more flexible, despite the expected performance hit. The reasoning beforehand was that if bottlenecks would arise, they could be dealt with on an individual basis. Recent testing and optimization has indicated that performance is not seriously affected by these decisions when the system is compared with competing systems.

3.3.2 Maintainability.

The TetraNode system has been under development for four years. The software system is expected to be between 100 KLOC (kilo lines of code) and 150 KLOC when it is finished. Currently the main components of the system are the TetraNode

Foundation Classes (TFC), a framework of reusable code of about 10 KLOC and the Protocol implementation (there are more components but we will limit ourselves to these). The TFC has been finished for a while and has been in maintenance since. The rest of the system is still being developed, so it is too early to make statements about the maintainability of the system. However, judging from code samples we have seen, the library code is reused quite effectively in the various protocol implementations.

Asked about bad practices, the interviewees told us that they avoided the use of inheritance and preferred to use templates instead. Also the use of pointers was restricted to object referencing (thus preventing memory leaks). We were also given access to an extensive code guidelines document detailing how the code should be structured and how certain C++ language constructs were to be used. In order to preserve system quality, the chief architects of the software carefully review new code. The development team of Rohill is too small, however, to have a formal review process in place.

3.3.3 Concurrency/synchronization.

Much to our surprise (we had anticipated that this would be a crosscutting concern), the interviewees told us that most of the synchronization related code of the system was located in the TFC and consequently was not much of a concern when implementing the protocols. The other classes in the system merely use the library classes and templates and are therefore free from synchronization issues.

3.3.4 Flexibility.

Neither of the interviewees had read the GoF book on design patterns by Gamma et al. [7]. However, both of them were aware of the notion of a design pattern and recognized that they probably implemented a few in the system. While Rohill expects that the Tetra protocol will see little change in the future, Rohill made an effort to include a lot of flexibility. Particularly C++ templates have been used frequently to add genericity to the system. In addition, to keep the software as portable as possible, they avoided creating too much dependencies between modules and limited themselves to using only the STL (standard template library that is part of the ANSI C++ standard) and their own TFC. Consequently, they anticipate little trouble in porting their software to e.g. Linux in the future.

3.3.5 Reliability.

While Rohill does not employ techniques to improve reliability, they do ensure that the system is reliable by performing extensive tests. Commonly, test machines run for weeks on end simulating real world usage of the TetraNode system. Also, there are some properties of the architecture that have the side effect of improving reliability. The design decision to put traditionally complicated concerns such as synchronization in a relatively small, well-designed library, for instance, ensures that the rest of the system is relatively simple and easy to maintain. Consequently it is also reliable. A side effect of having a library of reusable code is that this reduces redundancy in the code and centralizes the more complex and error prone parts of the system in a relatively compact library ensuring that if bugs surface, they are fixed centrally.

Also contributing to the reliability of the system are code reviews and the coding guidelines that are used internally. These guidelines, among others, prohibit/limit the use of bad pointer arithmetic, preprocessor directives and stimulate the use of templates and other advanced C++ constructs.

3.4 Summary

Rohill identified several concerns when they were designing their system and have optimized their software design for flexibility in those concerns.

First of all hardware portability is important to them so they have limited themselves to using ANSI C++ and only the standard C++ libraries (STL). This allows them to recompile on any platform with an ANSI compliant compiler. The benefit of this decision is twofold: first they can deploy on multiple platforms and second they can develop on e.g. MS Windows and benefit from the availability of sophisticated development tools and then deploy the compiled software to the target platform (e.g. VxWorks).

A second concern is the management of data packets used by the various protocol stacks. The designers of the system recognized early on that while the exact bit format of those packets varied from protocol to protocol, the information stored in the packets was more or less similar. Rather than reinventing this functionality for each protocol, Rohill created a framework for dealing with the packets.

4 Case 2: Vertis

Vertis is a IT services company that, among others, customizes, installs and maintains information systems primarily based on Oracle's database and tools. Typically this means that most of the software development is done in Oracle Designer, a tool for building database applications.

4.1 Application Domain

While Vertis has a great deal of expertise in database products and related services, the persons we interviewed were mainly involved with projects that make use of Oracle Designer. Oracle Designer is a tool that allows developers to specify application logic and user interface for database applications. The tool then automatically generates the necessary code to execute the application on top of an Oracle Database. Typically the tool is used to create administrative applications that are generally specific for the company they are developed for (i.e. they are one of a kind applications).

4.2 Development Method

The development method at Vertis can be characterized as following the waterfall model. Even though Vertis is increasingly adopting an iterative method called DSDM (Dynamic System Development Method), the tools and documents are still based on the old style of development. The process can be summarized as follows:

- First requirements are collected. Depending on the size of the project, the requirements are documented in more detail. Also the initial documents are often used as a basis for e.g. contracts.
- Based on the requirements, developers start specifying the databases schemas, design the user interface and define the application logic.
- After these artifacts have been defined Oracle designer combines the defined artifacts with predefined artifacts (e.g. Headstart, a set of predefined artifacts, from Oracle Consulting Services is used extensively) and generates an executable product.

The process is driven by a set of template documents that are part of the Oracle Custom Development Method (CDM). The document templates cover the whole software development process from requirements collection until deployment. The individual documents are considered to be deliverables of the various phases a project goes through. Depending on project size these templates are filled in more or less detail. For larger projects it is common that deliverables go through an extensive review process before being delivered to the customer.

4.3 Quality attributes

Unlike Rohill, Vertis works on a per project base. Rather than highlighting a concrete project within Vertis, we interviewed the interviewees about how the quality attributes are dealt with in general.

4.3.1 Performance.

Typically no explicit performance requirements are set on a project. However, there are some implicit performance requirements. Typically Vertis applications are interactive database applications and it is well understood that, for instance, in general queries should take only a few seconds at most.

In most cases the performance bottleneck is the database performance. Troublesome queries can be spotted using e.g. Oracle tools for analyzing queries and database schemas. If necessary, adjustments are made to optimize either database queries or schemas for the desired performance. A major problem in this area is that performance optimizations such as database indexes that are used to address performance problems may affect other parts of the system negatively. Consequently, performance tuning can be quite complicated and involves a lot of testing. However, since the frequency of such problems is relatively low, performance is only considered explicitly when problems are identified.

4.3.2 Maintainability.

Maintainability is increasingly important to Vertis. One of the interviewees indicated that the majority of development effort goes into maintenance of existing system. Less than half of the development concerns building new systems.

When developing new applications using Oracle Designer, developers try to avoid having to edit generated application code even though this is initially more expensive than editing the generated code by hand. In the past, editing generated code was often used as a quick way to fix little bugs and add minor features. Unfortunately, doing so also prevents that changes are made to the design without the post generation changes being lost.

Currently, Vertis is preparing a transition to a newer version of Oracle Designer that is able to generate web based applications rather than the traditional client server applications. Little trouble in the conversion process is expected with applications that are 100% generated since most of the old designs can be reused in the new version of Oracle Designer, allowing for an easy re-generation of the applications. Thus, a key design decision that enables this smooth transition has been to never edit the generated code.

4.3.3 Concurrency/synchronization.

While enterprise systems such as delivered by Vertis are generally distributed-, multi-tier systems, the complexity of handling the complexity of dealing with such system is fully encapsulated by the frameworks and tools used by Vertis.

4.3.4 Flexibility.

The generative approach used by Vertis ensures that radical changes to the system can be made by changing the generator. The prime example of this is the transition from client server based applications to web applications in the newer versions of Oracle Designer. While the generative approach is very flexible, Vertis is not in control of the generator. So it needs additional means of getting flexibility in its systems. One way of doing so is to carefully design the databases schemas to allow for e.g. additional fields. Another way of adding more flexibility is to make the data objects more generic to allow for e.g. run-time flexibility.

4.3.5 Reliability.

While reliability is an issue in enterprise systems, it is mostly taken care of in the Oracle tools and products. Additional measures Vertis takes to improve reliability are enforcing code guidelines provided by Oracle’s CDM method and performing code reviews. The code reviews can be quite rigid, especially in larger projects.

4.4 Summary

One would expect that the quality attributes we selected are very relevant in the domain of business applications. Interestingly, Vertis has managed to separate its primary concern, application logic, from the concerns we targeted in this study. By using and relying on application frameworks provided by third parties, Vertis has largely avoided that their products are tangled with e.g. performance related code or complex synchronization code. Our main conclusion for the Vertis case is that the domain in which they operate is so mature that the framework and tools they use provide adequate support for meeting the quality requirements of their projects.

5 Analysis

In this section we provide an analysis of our experiences in both cases. In our analysis we make a distinction between anticipated concerns and unanticipated concerns. The difference is that based on the requirements that are known in advance and the developer's experience with implementing applications in the domain, a set of concerns can be anticipated that are likely need to be addressed. We have found that in both domains we examined in our case study, such concerns exist. The remaining concerns are discovered later in the development cycle (e.g. during maintenance) and are therefore referred to as unanticipated.

For both types we discuss a number of examples in the context of the case studies. Further more, we look at the typical design solutions used to achieve separation of concerns. Finally, we also reflect on the usefulness of the application of Advanced Separation of Concerns (ASOC) technologies such as Aspect Oriented Programming and Subject Oriented Programming in these cases.

5.1 Anticipated Concerns

Anticipated concerns are identified early in the development process (e.g. during requirements analysis or simply because developers know from previous experience that a concern may become problematic if they don't do anything about it).

5.1.1 Examples

For Rohill we have been able to identify three different anticipated concerns that have been separated:

` **Throughput**. To have maximum throughput, an effort is made not to duplicate data packages that flow through the protocol stack. This design decision has a profound effect on the architecture.

` **Protocol implementation**. Most of the system's implementation is a straightforward implementation of the Tetra protocol specification. A sophisticated, C++ template-based architecture for implementing protocols is used. This architecture separates the functionality of the individual protocols from the generic functionality of managing finite state machines.

` **Platform independence**. To stay platform independent, the TFC contains a set of so-called OS-wrappers that abstract from platform OS-specific implementations of various functionality needed in the TetraNode system.

In the Vertis case, we have found that the following concerns have been separated:

` **All concerns addressed by the generation tool**. The generation tool encapsulates many concerns such as synchronization, security, network communication etc. While these concerns may still be mixed within the tool, this is of no concern to Vertis since Oracle is responsible for its implementation. The decision to use this product effectively separates all concerns addressed by this tool from the implementation of Vertis systems (assuming 100% generation of the systems).

` **Application logic**. One of the primary concerns that needs to be addressed by Vertis is implementing the functionality. Doing so is pretty much a straightforward conversion of the functional requirements to code. This is a good indication that the application logic concern is well separated from quality requirement related concerns such as e.g. synchronization.

- **Database performance**. Aside from the functionality, a second concern is the database schema design and optimization for certain queries. While tools exist to assist in getting optimal performance, it is very much a matter of manually fine tuning the system.

5.1.2 Traditional design solutions

Sophisticated design solutions are available for achieving the kind of separation of concerns needed in both cases. Within Rohill, a combination of two design solutions is used to achieve the desired separation of concerns:

- **Object Oriented Frameworks**. The TetraNode Foundation classes provide wrappers for OS functionality and base classes that lay out the systems architecture.
- **C++ templates**. In addition to object oriented frameworks, C++ templates are used to provide generic, reusable behavior for certain aspects of the system. A good example of this is manipulating the data packets that flow through the various protocol stacks. Often there are small differences between the ways these protocols data format. However, using the templates, protocol implementers need not worry about these differences.

Within Vertis, the architecture of the system is dominated by Oracle Designer. Consequently, there are no Vertis specific design solutions.

- **Frameworks + generator**. Oracle designer provides application frameworks and tool support for implementing the type of applications Vertis creates. The main difference between the Oracle Designer and e.g. the Vertis framework is that the frameworks provided by Oracle are more mature. In [19] a few different framework patterns are identified. In terms of these patterns the Oracle framework is much more advanced since, in addition to the reusable architecture, also a set of reusable components and a high level configuration tool are provided.
- **Code generation**. Another thing the Oracle Designer does is simplify the integration of the various artifacts included with the tool.

5.1.3 ASOC design solutions

Neither company uses ASOC solutions, so we will focus on how they could have used such solutions instead. Considering how both Rohill and Vertis make use of conventional techniques in order to separate concerns in their respective domains and how successful they are in doing so, we don’t expect that they would derive much benefit from using advanced separation of concerns techniques such as e.g. aspect oriented programming or subject oriented programming in the context of their existing systems.

In the case of Rohill, the concerns that were anticipated have been separated out into the TFC (TetraNode Foundation Classes). While theoretically, the design of this TFC could be improved by using more advanced techniques, this arguably would not help much since the TFC is relatively small compared to the rest of the system.

Vertis would not derive much benefit from using ASOC technology either. Vertis simply delegates any concerns related to e.g. quality requirements to the Oracle Designer tool. Of course Oracle might find that ASOC technology would be useful for implementing this tool. However, Vertis is not concerned with the tool’s implementation but only with using the tool to implement application logic.

5.2 Unanticipated Concerns

Unanticipated concerns typically become problematic during later stages of the development, for instance, because they are affected by new requirements. Typically, using e.g. refactoring techniques to separate such concerns has become more or less unfeasible by then due to the accumulated investment in the development of the system. Exploratory work, such as that of Murphy et al. [16], suggest that many concerns surface during development rather than that they are anticipated up front. However in our experience this may also indicative of an immature domain or a lack of domain understanding.

5.2.1 Examples

In the case of Rohill, we have to hypothesize about unanticipated concerns since the system is still under development. Consequently, any unanticipated concerns have yet to surface. Based on our own analysis, we have found that the following concerns may become problematic in the future under certain conditions (e.g. a change in requirements):

- **Design decision**. The architecture of Rohill’s TetraNode software is essentially a large finite state machine (representing the TetraNode protocol) implemented using C++ templates. While this appears to be a sound design decision (considering the size of the Tetra specification and the Rohill’s relatively small implementation of it), any decision to change the design might have system wide impact.
- **Billing**. In an earlier case study [21], we examined a system of a large telecommunications company that took care of billing customers for services on a telecommunication network. Changes in the billing concern caused major changes in the architecture for that system. While such billing functionality is unlikely to be required of the TetraNode system (even thought there is some rudimentary functionality and hooks for it in the system), it might very well cause some problems if it did since billing typically crosscuts the system. Also it is a good example of a change in concerns. Right now billing is not so important since TetraNode networks are not used for public networks.

As mentioned earlier, most of Vertis’ development concerns maintenance of earlier projects. Such maintenance activities involve adding functionality and adding new functionality to existing systems. Consequently, the unanticipated concerns have to be looked for in this area:

- **Unsupported functionality**. Since Vertis uses a code generator to generate applications, it is important that they keep changes restricted to editing the pre-generation artifacts such as predefined database forms and functionality. However doing so also prevents that functionality not supported by the generator is used. Until a few years ago it was common to make changes in the generated code since doing so is generally much easier for smaller changes. However, doing so also prevents editing the pre-generation artifacts since that requires reverse engineering the changed code. Consequently maintenance cost of these artifacts is much higher compared to a situation where the generated code is not edited.

- **User interface**. Three generations of Oracle designer can be distinguished. The first generation created text based database applications, the second generation generated GUI based applications and the latest generation can also generate web-based applications. Since Vertis tries to prevent post generation editing of the systems, one would expect that by simply upgrading Oracle Designer, new applications can be generated. For the last generation this is indeed more or less true, however the transition from the first to the second generation was less smooth due to the fact that this transition occurred before the decision not to do post generation editing was made and because the design of the user interface contained things that were specific for text based interfaces.

5.2.2 Traditional design solutions

While Rohill has made an effort to make their design flexible, it is simply impossible to anticipate all future requirements. When unanticipated concerns need to be incorporated the following techniques can be used:

- **Refactoring**. As discussed in [16], it is possible to separate a concern from other concerns in an existing OO system using conventional OO techniques (Murphy et al. use a process based on lexically analyzing source code with tools like for instance grep). Typically, the process involves refactoring and restructuring the code. Using these design techniques, software architects can optimize their designs for concerns. However, applying such techniques later in the development process can be costly. Aside from the cost factor, the refactored system may be incompatible with the old system so any depending systems need to be updated as well.
- **Compromise the design**. When redesigning is no option (e.g. because of compatibility reasons), developers may choose to add new functionality while preserving the original design as much as possible. Generally this increases complexity of the system significantly and ultimately it may lead to design erosion (also see [9]).

In Vertis, addressing unanticipated concerns may be hard to address since essentially, the architecture of the system is limited by what the Oracle tool generates. Consequently they only have one option:

- **Post generation editing**. When functionality that is not supported by Oracle Designer is needed, it can be added by editing the generated code. Of course there is no guarantee that the new functionality fits in nicely with the generated code so potentially, refactoring of the generated code is needed as well. Since the Oracle Designer tool has improved substantially over the last few years, post generation editing of code is generally discouraged within Vertis because of the higher maintenance cost (even though initial development is considered to be cheaper).
- **Not using the generator**. This way all the functionality that is normally generated needs to be implemented as well. In most cases it therefore is not a feasible option.

5.2.3 ASOC design solutions

Techniques such as Aspect Oriented Programming could be used to implement unanticipated concerns. However, as discussed in [16], the usage of e.g. AspectJ (an aspect oriented version of Java) or HyperJ (a subject oriented programming) usually requires that the original program is adjusted as well. Consequently, it cannot be assumed that these techniques can be applied to add new concerns to an existing system without requiring that the original system is changed as well.

- **Billing aspect**. In the case of Rohill, using e.g. Aspect Oriented Programming would seem ideal to implement the billing concern. Billing is a good example of a cross cutting concern since it has a relatively well defined set of functionality that needs to interact with a system in multiple places (in AspectJ such places would be referred to as joinpoints). However, it is uncertain that such joinpoints are readily available in the TetraNode system. The work by Murphy et al. [16] suggests that refactoring the system to obtain these joinpoints might be necessary.
- **Apply ASOC to generated code**. In Vertis, the ASOC technologies can only be applied to the generated code. Unfortunately, such a thing would most likely require editing the generated code manually so it is not a very attractive option.

5.3 Summary

Table 1 Design solutions for anticipated and unanticipated concerns

SOC	Anticipated Concerns	Unanticipated Concerns
Conventional techniques	The systems can be designed to handle anticipated concerns well.	Potentially, a lot of refactoring is needed.
ASOC techniques	Not needed when conventional solutions are available but may improve the flexibility of the design.	Of limited use due to the need to change the existing system. However, it may be a better option than using conventional techniques.

In Table 1, we have summarized our analysis. While the two domains we included in our case study are very different, the analysis shows that our conclusions can be generalized. When concerns are anticipated, a system’s design is optimized for these concerns in such a way that these concerns are separated. In both cases we have observed that conventional techniques allowed for good separation of the known concerns. In one of the cases a generator was commonly used to compose the remaining concerns with the other concerns.

Typically, using ASOC techniques for anticipated concerns may improve the design, however conventional techniques do the job well enough that these techniques are not really needed. With unanticipated concerns, extensive refactoring may be needed to achieve separation of concerns using conventional techniques. Using ASOC technology may help but even then, laborious refactoring can probably not be avoided.

Exploratory studies such as [16], suggest that many concerns surface during development rather than that they are anticipated up front. While this case study somewhat contradicts this claim since in both cases we have found relatively mature systems with a good level of separation of concerns, it is also very clear that most concern related problems arise from unanticipated concerns rather than anticipated concerns. Therefore we believe that future research in the ASOC community should focus on providing support for the separation of unanticipated concerns.

Meanwhile, adopting ASOC technologies does not seem to have a negative impact. Lippert and Lopes [13] conclude that “The worst case scenario with aspects is not much worse than the original implementation“ in their case study. Consequently, a case can be made for opportunistically adopting e.g. Aspect Oriented Programming in order to derive the benefits at a later stage.

6 Limitations of this study

In this section we will discuss the validity of the case study and discuss the way it was executed. Our case study consists of four structured interviews at two companies from two different domains.

- **Validity of the answers.** In each company we interviewed two persons. This allowed us to compare their answers and resolve inconsistencies between the answers during the feedback phase. In addition, if both interviewees give the same answers, that is a good indication of the validity of the answer.
- **Minimizing problems.** When discussing quality issues there is a risk that developers tend to be reluctant to admit there are quality issues. Consequently, they may downplay relevant issues. However, in both cases we interviewed experienced, senior developers and we are confident that their answers were accurate.
- **Disjunct domains.** The domains in which the two companies operate are very different. This means that any conclusions we can generalize for these two cases are likely applicable to other domains and companies as well. However, additional case-studies are needed to confirm such conclusions.
- **Representative cases.** Even though we believe otherwise, there is a risk that these companies may not be representative for their respective domains and that some of the conclusions we have drawn cannot be generalized.
- **Knowledge gap.** Doing an ASOC casestudy by interviewing software engineers has the limitation that generally, software engineers have no knowledge of the related terminology since most ASOC material available is of a rather academic nature. We bridged the gap in knowledge by discussing in terms of quality attributes rather than concerns. This approach introduces a necessary level of indirection and also requires an additional interpretation step in order to relate the interviews to separation of concerns. Training the interviewees would solve this issue. However, in addition to the fact that this would take a lot of effort it also has the side effect of influencing the interviewees, which would make it harder to generalize our conclusions.
- **Number of quality attributes.** We limited the interviews to five quality attributes. Consequently we may have overlooked important concerns that affect other quality attributes (e.g. security). Adding more quality attributes would have required us to either have more/longer interviews or reduce the amount of time for discussing each concern. We feel that given the two hour time slot reserved for the interviews this is the best we could do.

The methodology we applied to this case study is well suited considering the time constraints. It allows for comparing the results of each case because of the interview structure. Yet, it also allows us to extract case specific information. Unfortunately, our methodology also has some inherent limitations that limit the scope of our conclusions. The uncertainty of our results resulting from these limitations can be addressed by conducting more extensive case studies in other domains.

7 Related Work

7.1 Separation of Concerns

By separating crosscutting concerns at the implementation level, the effect of changes affecting only a particular concern can be isolated. E.g. by separating the concern synchronization from the rest of the system, changes in the synchronization code will not affect the rest of the system. Examples of approaches that aim to improve separation of concerns are Composition Filters [1], Aspect Oriented Programming [12], Subject Oriented Programming [10] and Multi Dimensional Separation of Concerns [23]. An issue with these approaches is that these are mostly implementation level approaches. While some approaches for using ASOC on the detailed design level have been suggested (e.g. [4]), good design level equivalents of the concepts used in the implementation level ASOC techniques are currently lacking. Consequently not much benefit is derived from using them for anticipated concerns since typically such concerns can already be addressed using conventional design techniques. Since using such techniques results in a system that is not explicitly prepared for deploying ASOC techniques, the use of such techniques for implementing unanticipated concerns is limited too since typically this requires that the system is refactored in some way [16].

7.2 Design techniques

In both cases, OO Frameworks are used. Roberts and Johnson [19] describe a few basic concepts and related terminology. Also they present a pattern language that suggests how a framework approach can be adopted. In [8] we describe a few guidelines that can help improve the flexibility and maintainability of OO frameworks.

In the absence of mature separation of concerns techniques, developers can resort to introducing variability into their software architectures. Fowler et al. [6] present an overview of techniques that can be used to refactor OO systems. Such techniques can be used to use design patterns such as described by Gamma et al. [7] for architectural patterns such as described by Buschmann et al. [3].

7.3 ASOC case studies

Murphy et al. [16], present a study that describes how to medium sized OO programs were re-architected using three different separation of concerns techniques: AspectJ, HyperJ and an ad hoc method invented by one of the authors called lightweight concern separation. Important conclusions of this work is that applying such techniques to an existing OO system cannot be done cleanly in most cases (i.e. it is necessary to refactor the system to apply these techniques).

In [14], a case study about the application of an aspect oriented version of CLOS to an image processing software package. However, this case study is explicitly focused on demonstrating the capabilities of AOP and the example involved is an academic case rather than an industrial one such as our case. In another AOP case study by Lippert and Lopes [13], an existing OO system is refactored. Interestingly this leads to a substantial reduction in size. However it should be noted that the OO system examined is relatively small (approximately 44,000 lines of code) compared to our cases and that the amount of work needed to refactor the system is probably substantial.

Finally, Kersten and Murphy [11] present an AOP case study that focusses on the practical side of adopting an AOP approach. One of the lessons learned in this paper is that proper design constructs for aspects would be useful and that one should be careful not to make the code to complex.

7.4 Other work

In earlier case studies [14][21] and [9] we have found that software designs tend to erode over time. Due to the fact that incremental changes are not properly designed the system increasingly becomes less prepared for future incremental changes. Systems that were optimally designed for anticipated concerns are gradually refactored to deal with new requirements also affecting unanticipated concerns. As this study argues and as is confirmed in the study by Murphy et al. [16], simply applying ASOC technology to existing systems in order to separate these concerns potentially involves substantial refactoring of the original system.

8 Summary

In this paper we present the results of a case study we conducted at two local SMEs in two domains. The goal of our study was to find out how concerns are separated in industrial products. A secondary goal was to research how ASOC technology could be used to improve things. Due to the inherent limitations of a case study, we limited ourselves to conducting structured interviews. A discussion of this method can be found in Section 2 and Section 6. Our conclusions of this case study can be summarized as follows:

- In both cases we have found that effective design solutions were used to separate the so-called anticipated concerns in such a way that expected future requirements can be met with little effort. Consequently, the use of ASOC technology to separate such concerns may be useful but is not likely to improve separation of concerns much.
- The systems are not prepared for unanticipated concerns that may, for instance, be affected by unexpected requirements. In the case of Rohill it is too early to identify such concerns since the system is still under development. In the Vertis case, however, experience shows that unexpected requirements that affect unanticipated

concerns in the system, that are not addressed effectively by the tool they are using, may pose serious problems.

` Existing work (e.g. [16]) suggests that applying ASOC technologies to existing OO systems may potentially require that the system is refactored. Consequently we conclude that using such technologies in the cases we discuss in this paper would not be of much help in separating the unanticipated concerns.

While we are confident these conclusions are valid for both cases, only additional, more extensive studies may show that these conclusions can be generalized. However, the conclusions are in line with our experiences with previous cases (e.g [9][14][21]). In our discussion of the research method (See Section 6), we have listed a number of issues with the method. In order to address these issues, further case studies are needed.

We are currently considering conducting a survey among senior level software engineers at various companies in order to find out whether our conclusions are valid for a wider range of domains. By sending this survey to a large group of people, we may be able to further generalize our conclusions. In addition, repeating this case study in a more extensive form (e.g. by including more quality attributes) may enhance our conclusions.

9 Acknowledgements

We would like to thank Rohill B.V. and Vertis B.V. for their participation in our case study and their invaluable feedback on the results.

10 References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, A. Yonezawa, "Abstracting Object Interactions Using Composition Filters", Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming, Springer-Verlag, pp. 152-184, 1994.

[2] Aspectj, "The AspectJ Programming Guide", http://aspectj.org/servlets/usersGuides/prog/progguide.htm, last verified 20 August 2001.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, "Pattern-Oriented Software Architecture: A System of Patterns", John Wiley & Sons, 1996.

[4] S. Clarke, W. Harrison, H. Oscher, P. Tarr, "Subject Oriented Design: Towards Improved Alignment of Requirements, Design and Code", OOPSLA '99.

[5] ETSI homepage, http://www.etsi.org/.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, "Refactoring - Improving the Design of Existing Code", Addison Wesley, 1999.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns - Elements of Reusable Object Oriented software". Addison-Wesley, 1995.

[8] J. van Gurp, J. Bosch, "Design, Implementation and Evolution of Object Oriented Frameworks: Concepts & Guidelines", Software Practice & Experience no 33(3), pp 277-300, March 2001.

[9] J. van Gurp, J. Bosch, "Design Erosion: Problems & Causes", submitted May 2001.

[10] W. Harrison, H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", Proceedings of OOPSLA '93, pp 411-428.

[11] M. Kersten, G. Murphy, “Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-oriented Programming“, Proceedings of OOPSLA ‘99.
[12] G. Kiczalez, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, “Aspect Oriented Programming“, Proceedings of ECOOP 1997, pp. 220-242.
[13] M. Lippert, C. Lopes, “A study on exception detection and handling using aspect oriented programming“, Proceedings of ICSE 2000.
[14] M. Mattsson, J. Bosch, “Framework Composition Problems, Causes and Solutions”, Proceedings Technology of Object-Oriented Languages and Systems, USA, August 1997.
[15] A. Mendhekar, G. Kiczales, and J. Lamping. “RG: A case study for aspect-oriented programming”, Technical Report SPL97-009P9710044, Xerox PARC, Feb. 1997
[16] G. C. Murphy, A. Lai, R. J. Walker, M. P. Robillard, “Separating Features in Source Code: An Exploratory Study“, Proceedings of ICSE 2001, pp. 275-284.
[17] G. C. Murphy, R. J. Walker, E. L. A. Baniassad, “Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-oriented Programming“, IEEE Transactions on Software Engineering, 25(4):438--455, July/August 1999.
[18] W. Pree, K. Koskimies, “Rearchitecting Legacy systems - Concepts and Case study”, First Working IFIP Conference on Software Architecture (WICSA '99), pp. 51-61, February 1999.
[19] D. Roberts, R. Johnson, “Patterns for Evolving Frameworks”, Pattern Languages of Program Design, vol 3, pp. 471-486, Addison-Wesley, 1998.
[20] Rohill website, http://www.rohill.nl/.
[21] M. Svahnberg, J. Bosch, “Evolution in Software Product Lines: Two Cases”, Journal of Software Maintenance, Vol. 11, No. 6, pp. 391-422, 1999.
[22] M. Svahnberg, J. van Gurp, J. Bosch, “A Taxonomy of Variability Realization Techniques“, submitted June 2001.
[23] P. Tarr, H. Ossher, W. Harrison, “N Degrees of Separation: Multi-Dimensional Separation of Concerns“, Proceedings of ICSE’99, pp. 107-119.
[24] Vertis website, http://www.vertis.nl/.