Why Software Engineers Do Not Keep to the Principle of Separating Business Logic from Display: A Method Rationale Analysis*

Malin Häggmark and Pär J. Ågerfalk

Dept of Computer Science and Information Systems, University of Limerick, Limerick Ireland; and Dept of Informatics (ESI), Örebro University, SE-701 82 Örebro, Sweden
malin.haggmark@ul.ie, par.agerfalk@ul.ie

Abstract. This paper presents an investigation into why software engineers do not keep to the principle of separating business logic from display. The concept of method rationale is used to establish what is supposed to be achieved by following the principle. The resulting model is then contrasted with results from in-depth interviews with practicing engineers about what they want to achieve. The difference between what the principle advocates and what engineers consider beneficial holds the answer to why the principle of separating business logic from display is not maintained. The results suggest that many espoused benefits of the principle do not appeal to engineers in practice and the principle is tailored to make it more useful in particular contexts. Tailoring the principle also brought about other benefits, not explicated by the principle, thus reinforcing the idea that method tailoring is crucial to the successful enactment of information systems engineering methods.

1 Introduction

The mantra of most experienced software engineers is the same: *thou shalt separate business logic from display* [19]. Theory maintains that by separating business logic from display, systems will be easier to scale, extend, update and maintain. Also, engineers with different skill sets can work on different parts of a system independently, thus optimizing tasks for each competence. This way of structuring systems, typically with business logic and display structured in different tiers, also facilitates new types of clients to be added with little extra effort [9, 14, 16, 19, 23].

The principle of separating business logic from display can be found in a wide range of information systems engineering (ISE) methods. In general, methods are used in ISE as a means of expressing and communicating knowledge about good (effective and efficient) ISE practice. This way methods encapsulate knowledge of good engineering practice, and by utilizing this, engineers can be more effective, efficient and confident in their work [2]. Basically, a method is a proposed pattern of

* This work has been financially supported by the Science Foundation Ireland Principal Investigator projects B4-STEP and Lero, and the EU FP6 Project COSPA.

© Springer-Verlag Berlin Heidelberg 2006

activities, expressed as a set of prescriptions for action – a.k.a. method prescriptions [3]. When the principle of separating business logic from display is part of an ISE method, keeping to this principle involves following a set of such prescriptions, i.e. following a method. Of course, this principle alone does not provide sufficient support for successful ISE. The point is rather that we can choose to view it as a method fragment [4] – as a set of method prescriptions – in order to draw on previous research on method use in ISE practice. Specifically, this research suggests that if engineers are to follow a method, the method must first and foremost be useful [22], that is, enable them to be more productive and achieve higher levels of performance in their job. For someone to regard a method as useful the knowledge must be possible to rationalize, i.e. the person needs to be able to make sense of it and incorporate it into their own view of the world [2]. It has been stressed that departure from methods is conscious and inevitable in the real world, and that rigorous use of a method does not pay back [6, 25]. Engineers tailor methods to suit their needs in particular situations with awareness of the benefits and drawbacks this causes [6].

By entangling business logic with display the development time may be shortened, but may disadvantageously result in a harder and more tedious maintenance process. This suggests that engineers emphasize short-term benefits [19]. Research has shown that students have difficulties learning how to structure applications [5], suggesting that engineers do not keep to the principle because they have not fully understood how to use it. A possible solution would be to enforce the principle of separation in, for example, the template engine; leaving no possibility to entangle business logic with display [19]. It can be questioned whether enforcement is appropriate, and hence a more thorough investigation of *why* engineers do not keep to the principle is necessary.

To summarize: A method (or any of its parts) has to be useful for engineers to keep to it [22]. Furthermore, engineers must be given the freedom to tailor the method to make it useful in their particular situation [6, 25]. Engineers may not tailor methods in a way that is beneficial for the software produced, but rather to ease and speed up the process of software development, thus, deteriorating the quality of the product (the software) in favour of the personal process goals [19]. Their rationale is rarely explained; they make design decisions with no clear statement of why they do things the way they do [18]. The principle of separating business logic from display does not seem to be an exception from the rule that methods need to be tailored. The principle is espoused as ideal in theory, but practice seems to be a different story altogether [19]. This paper is an empirical enquiry into why engineers do not keep to the principle of separating business logic from display. Answering this question also increases our understanding of the more fundamental question of how and why engineers choose to tailor ISE methods in general.

The contribution of this paper is thus threefold. First, the essence of the principle of separating business logic from display is captured and expressed as a set of method prescriptions. Second, why software engineers do not keep to this principle is investigated. Finally, the usefulness of method rationale as an analytic tool to understand method tailoring is explored.

The paper proceeds as follows. Section 2 covers rationality of methods, and how this can be used to analyse methods. It sets the foundation for the research method, outlined in Section 3. The result of the investigation is presented and analysed in Section 4, preceded by conclusions in Section 5.

2 Methods and Their Rationale

A method is always grounded in a way of thinking [7, 10, 12], which constitutes the foundation for the reasons and arguments behind it. These reasons and arguments can be referred to as *method rationale* [1, 2, 24]. Using method rationale to understand the enactment of ISE methods can be facilitated by discussing it in terms of public and private rationality [26].

Public rationality is about creating an inter-subjective understanding (about the reasons and arguments) of the method. This is a sort of knowledge that is shared by several people as part of their inter-subjective beliefs. Public rationality can be externalized and communicated through written method descriptions (e.g. a method handbook). Public rationality is expressed in an *ideal typical method* [2] or *method-in-concept* [15]. Private rationality, on the other hand, is personal and cannot be externalized in every respect. Private rationality can be found in a person's 'skills and professional ethical and aesthetic judgements' [26]. Private rationality is expressed in a *method-in-action* [2, 15]. In an ideal situation, public and private rationality fully overlap [13]. If so, the method prescription can be carried out to a tee since the engineer fully understands and agrees with everything suggested by the method. This overlap, referred to as *rationality resonance* [26], is depicted in Fig. 1.

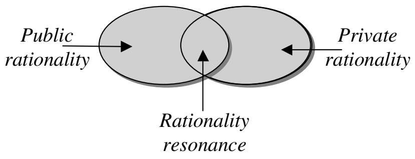

Fig. 1. Rationality resonance [13]

The non-overlap gives rise to a *method usage tension* [15] – a tension between what ought to be done (according to the method creator) and what is actually done. Analysing rationality resonance requires that both private and public rationality are made as explicit as possible to enable comparison [13]. Fig. 2 provides a visual overview of method rationale by showing how its constituent concepts relate.

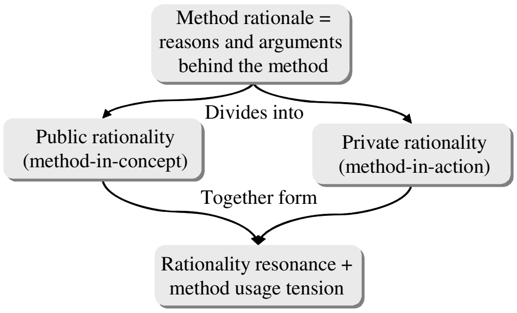
Method rationale = reasons and arguments behind the method
Divides into
Public rationality (method-in-concept)
Private rationality (method-in-action)
Together form
Rationality resonance + method usage tension

Fig. 2. The constituents of method rationale

2.1 Public Rationality Analysis Through Goals and Values

A methods' public rationality can be analysed with respect to the goals and values it implements. The reasons behind a method prescription can be understood in terms of the goals the prescription is supposed to realize. This way, each method prescription can be related to one or more goals, even though these are not always well articulated in the method descriptions. A goal can be defined as a result, towards which behaviour is consciously or unconsciously directed [3].

Ultimately, public rationality lies in the heads of the people who have developed a method [24]. Accordingly, goals are manifestations of the method creator's value base – all goals are anchored in values. A value can be understood as an ethical judgement like an expression of feeling and attitude and can therefore not be judged as true or false. Goals can be related to each other in goal hierarchies; for example, when a goal is as a means to achieve another (higher) goal. Similarly, values can be anchored in other values. These two properties of method rationale are referred to as *goal achievement* and *value anchoring*, respectively. In addition to goal achievement, there is a possibility that goals contradict rather than complement each other – hence there is an additional *goal contradiction* relation defined over the set of goals. Similarly there is a *value contradiction* relation defined over the set of values. Fig. 3 depicts how every method prescription is related to at least one goal, and each goal is related to at least one value. [3] See Section 4.1 for concrete examples.

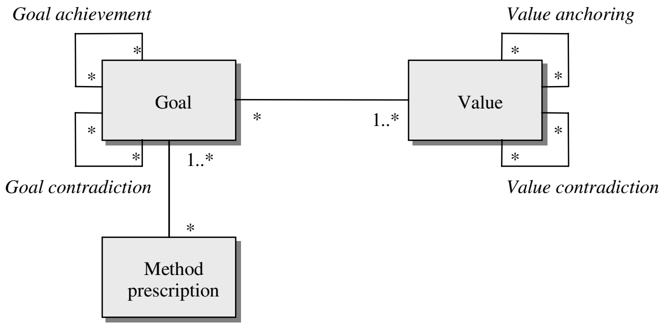

Fig. 3. Method rationale as constituted by goals, values and their relationships [3]

How an engineer chooses to use a particular method prescription depends on the goals this prescription helps to achieve. Whether or not a goal appeals to an engineer depends on whether or not they subscribe to the value in which the goal is anchored; i.e. whether or not rationality resonance can be achieved [3].

2.2 Modelling Public Rationality

The directed graph in Fig. 4 gives a visual representation of how method prescriptions, goals and value are related. It shows that Goal 1 is achieved by following the Method Prescription. Goal 2 is a goal on a higher level, which Goal 1 is a means to achieve [11]. Goal 1 is anchored in Value 1 and Value 2, which in turn are anchored in Value 3 and Value 4. Goal 2 is anchored in Value 5.

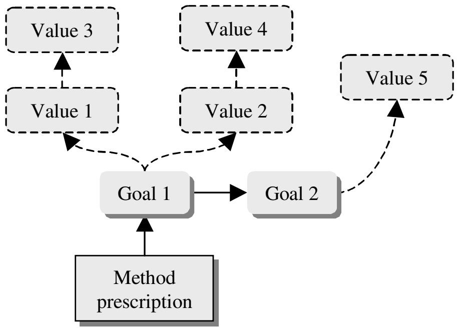

Fig. 4. Visual representation of public rationality through goals and values

3 Research Method

A qualitative research approach with structured interviews [20] was used in this research. A visual presentation of the adopted research approach is shown in Fig. 5 and explored in the remainder of this section.

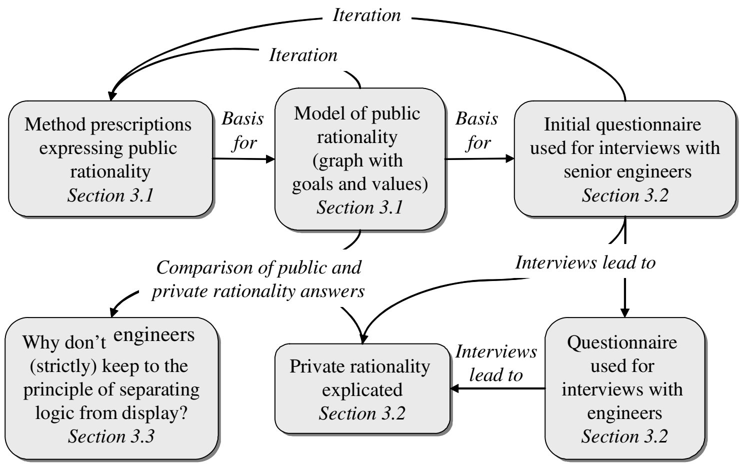
Iteration
Iteration
Method prescriptions expressing public rationality Section 3.1
Basis for
Model of public rationality (graph with goals and values) Section 3.1
Basis for
Initial questionnaire used for interviews with senior engineers Section 3.2
Comparison of public and private rationality answers
Interviews lead to
Why don't engineers (strictly) keep to the principle of separating logic from display? Section 3.3
Private rationality explicated Section 3.2
Interviews lead to
Questionnaire used for interviews with engineers Section 3.2

Fig. 5. Research design

3.1 Capturing Public Rationality

As described above, expressed method prescriptions are the foundation for the ideal typical method-in-concept and as such are expressions of public rationality. By analysing these, the goals[1] and values that underpin them can be explicated. The analysis results in a graph, such as the one in Fig. 4. The method-in-concept is in our

[1] These are the goals of the method creator(s), which they aim to communicate through method prescriptions [2].

case rather generic since the principle of separating business logic from display does appear, as mentioned above, in many methods. Hence, we need to capture the essence of the principle by arriving at a synthesis from sources that represents frameworks widely used (see Section 4.1). This synthesis constitutes the foundation for elaborating prescriptions, goals, values and their interrelationships, arriving at a model depicting the public rationality (visualized in Fig. 6 and Fig. 7).

3.2 Capturing Private Rationality

The next step is then to test which of the 'public rationality values' that are in accordance with the engineers' values, thus elaborating the private rationality by identifying how they use the method prescriptions. Asking questions that capture those values illuminate the engineers' value base. As explained above, this value base is the foundation for their goals, so focusing the values during interviews will implicitly extend to the goals.

The goal-value-model is the input for designing a questionnaire used in the interviews. One or several questions capture each value in the model. For example, the value 'It is easier to locate and determine problems/bugs in applications composed of well demarcated parts', is captured by the question: 'Do you consider it easier or harder to track down problems/bugs when the application separates logic from display?' Repeating this step to cover each value results in a questionnaire suitable for structuring the interviews. To clarify which question(s) captures which value(s), a table, such as Table 1, is used[2]. The actual values identified are presented in Fig. 6 and Fig. 7 in Section 4.1.

Table 1. Values and the corresponding questions

Value	Question
V1	Q1-Q4 Q10
V2	Q5 Q8
V3	Q3-Q6
...	...

The private rationality was explicated by performing (and recording) semi-structured open-ended interviews [20, 28]. This approach gives the opportunity to get a focused, thorough, insightful understanding of the engineers' perspective on tier-based development[3], enabling the engineers to speak freely about their work. The selection of respondents was inspired by the ethnographic principle of selecting a representative individual for initial enquiry who then suggests further respondents. The number of respondents is then increased until saturation is achieved – that is, until the marginal utility of further interviews are deemed insignificant. Such an approach avoids researcher bias and allows for more objective results. Initially a group interview was carried out with two highly experienced senior engineers/project managers. The questionnaire was here used as a guide, but the aim of this interview was primarily to find weaknesses and improve it for further interviews. This interview

[2] For the complete questionnaire and value-question-table used in this research see http://www.csis.ul.ie/staff/paragerfalk/CAiSE2006-Q-V.pdf

[3] Separating business logic from is typically implemented by structuring the software into tiers.

resulted in a questionnaire with more sub-questions and less ambiguity. The improved questionnaire was then used to interview a total of five engineers individually. All respondents were notified of the study in advance, but did not get the questions beforehand. The interviews lasted for 30–50 minutes each.

3.3 Analysing Rationality Resonance

The differences in values (i.e. the method creators' values versus the engineers' values) lead to an understanding of why these engineers do not keep to the principle, thus revealing information about the *method usage tension* in terms of *rationality resonance*. Comparing each 'public rationality value' from the model with the 'private rationality values' from the interviews shows which values differ, thus answering the question why engineers do not follow the principle of separating business logic from display.

3.4 Organizational Context for the Interviews

The interviews took place in the IT department of Statistics Sweden – the Swedish public authority responsible for all official statistics. The IT department provides the organization with applications for gathering and processing data. Typically, data is gathered via web-based clients, which is later processed in windows-based clients.

The organization has previously mainly developed small systems. An application with 1–10 users (method statisticians) has been the most common application. Increasing demands for larger applications, and for integration of various systems, has led the organization to leave their regular Visual Basic environment in favour of the object oriented (OO) multi-tier based .NET-framework. The respondents were all experienced engineers with good knowledge of both environments.

4 Results and Analysis

This section gives a theoretical presentation of the principle of separating business logic from display, explicating the reasons behind the method prescriptions. The goals and values of the method prescriptions are then analysed and presented in Fig. 6 and Fig. 7, followed by the result from the interviews. All references to specific method prescriptions (P), goals (G) and values (V) in this section refer to those figures.

4.1 The Pros and Cons of Separating Business Logic from Display

A most straightforward way to develop a system could be to interweave the display with the business logic. This is probably not a bad idea if the application is relatively small, supports a single type of client, and is not expected to be considerably extended or updated. Dividing an application into different tiers will increase its complexity since extra classes will be required to handle the separation of display and business logic [14, 19, 23, 27].

The idea behind structuring a system in tiers is to achieve separation of concerns (G3, G6, G7, V3–V5, V7, V13); it is much more difficult to change the display if it depends on and is built into the business logic, and vice versa [8, 16, 21]. Separation of these areas of concern generally results in more flexible systems, with the ability to support multiple types of clients (G1–G2, V1–V2) [16, 17]. From this reasoning, the method prescriptions P1 and P2 become apparent.

Dividing the application into separate parts is a kind of encapsulation that enhances the manageability and maintenance [8, 9, 17]. Because each task is contained within its own object, it is easy to locate and determine where a problem exists (G3, V5) [14, 19]. Designers can develop/update the display without the need to contact programmers (V6–V10) [14, 19]. Thus, labour is divided according to different skill sets (G4), as recommended by P4. This encapsulation and breaking down of large tasks into smaller ones also provide for component reuse (G5, V2, V11–V13), either within the project, or in other similar projects, giving rise to P3.

In general, the benefits of tier-separation arise when [14, 19, 27]:

1. The application will support multiple types of clients. Since the display is separated, all that is needed is to create a new type of client, and let it access the business logic.
2. The business logic is likely to be updated or extended throughout its lifecycle.
3. The display is likely to be updated or extended throughout its lifecycle, for example with new ‘skins’, to improve the looks.
4. The development team develops/maintains more than one application; components can be reused between (but also within) projects.
5. The development team is composed of individuals with different skill sets.

Fig. 6 and Fig. 7 depict a graph representation of the goals and values of the principle of separating business logic from display and how they are related, i.e. it depicts an explicit model of the public rationality.

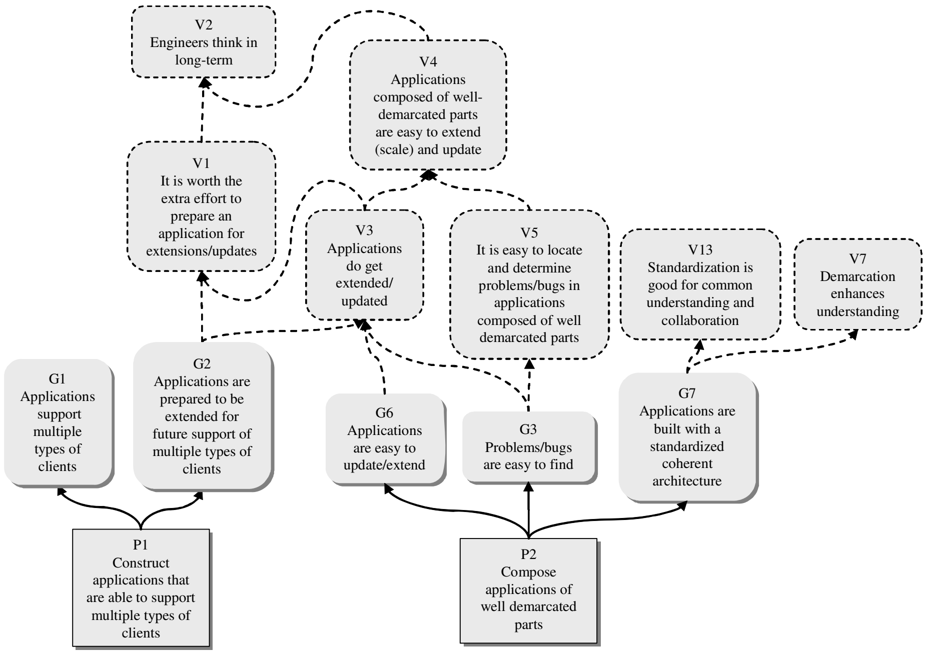
V2
Engineers think in long-term
V4
Applications composed of well-demarcated parts are easy to extend (scale) and update
V1
It is worth the extra effort to prepare an application for extensions/updates
V3
Applications do get extended/ updated
V5
It is easy to locate and determine problems/bugs in applications composed of well demarcated parts
V13
Standardization is good for common understanding and collaboration
V7
Demarcation enhances understanding
G1
Applications support multiple types of clients
G2
Applications are prepared to be extended for future support of multiple types of clients
G6
Applications are easy to update/extend
G3
Problems/bugs are easy to find
G7
Applications are built with a standardized coherent architecture
P1
Construct applications that are able to support multiple types of clients
P2
Compose applications of well demarcated parts

Fig. 6. Goals and values for the principle of separating business logic from display

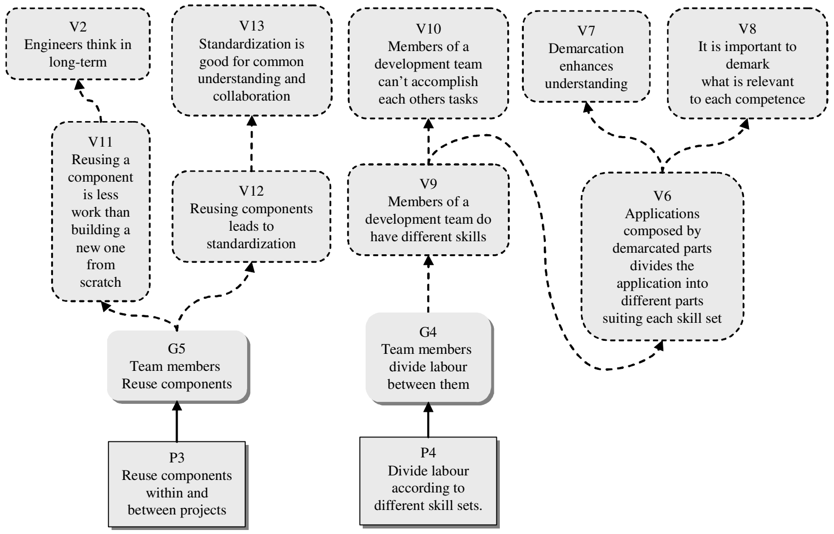
V2
Engineers think in long-term
V13
Standardization is good for common understanding and collaboration
V10
Members of a development team can't accomplish each others tasks
V7
Demarcation enhances understanding
V8
It is important to demark what is relevant to each competence
V11
Reusing a component is less work than building a new one from scratch
V12
Reusing components leads to standardization
V9
Members of a development team do have different skills
V6
Applications composed by demarcated parts divides the application into different parts suiting each skill set
G5
Team members Reuse components
G4
Team members divide labour between them
P3
Reuse components within and between projects
P4
Divide labour according to different skill sets.

Fig. 7. Goals and values for the principle of separating business logic from display (continued)

4.2 Results from Interviews – Public Versus Private Rationality

This section is a synthesis of the result from the interviews. It is structured along the four method prescriptions (see above) focusing differences and similarities (i.e. method usage tension and rationality resonance) of what is found in the interviews (private rationality) and what the theory/method-in-concept tells us (public rationality).

P1: Construct Applications that are Able to Support Multiple Types of Clients
The organization uses two types of clients, windows clients and web clients in their applications. It is always known beforehand if the application shall have windows clients, web clients or both; a system has never been extended with a new type of client afterwards. It is more common to build two applications instead of one application with two types of clients. They do reuse components in the two different applications. Their aim is to integrate applications, thus having different clients access the same business logic, but so far this has not been achieved. The following goal contradiction came up during the interviews:

'*Theoretically, it is possible just to add an extra client to an existing application. In reality though, you need to structure the application differently if it is a windows or web application. With a windows application you can do a lot more, for example, you can keep a big object with a lot of attributes in memory the whole time, there will always be enough memory for this. In a web application you need to be more careful with the resources (like memory), since it is on a server (perhaps a web hotel), and you don't know how many will be using it simultaneously. If you choose to adjust all windows applications for web use, these applications will be a bit "handicapped", and not as advanced as they could be. The windows application functionality will then not be fully utilized.*'

Security issues make it more complicated to add web clients to systems, since web clients are not allowed direct access to the 'inner' servers. Data has to be replicated on special 'outer' servers in these occasions.

As indicated above, the public rationality goals G1 (Applications support multiple types of clients) and G2 (Applications are prepared to be extended for future support of multiple types of clients) were not really the goals of the organization. It was more common to do a separate application for each client. This had partly to do with security issues. The engineers also claimed that an application has to be structured differently depending on what kind of client it will be used for; optimizing for one type gives drawbacks for another, etc. G1 and G2 are not achieved in this organization.

Value V1 (It is worth the extra effort to prepare an application for extensions/ updates), does not seem to hold the answer to why G1 and G2 is not fulfilled. The answer lies in what is described above about structuring applications differently depending on which types of clients it will have. The engineers pretty much say that (this part of) the theory is too good to be true; it is impossible to put into practice. It is beyond this investigation to analyse this further, but there is a possibility that the engineers had not fully understood the method-in-concept. V2 (Engineers think in long-term) is a succession of V1, and is therefore not possible to evaluate in this context. However, it will be touched upon in the next section. The value V3 (Applications do get extended/updated (in this case with additional types of clients)), is not consistent with the values in the organization, since the systems were not to be extended with new types of clients.

P2: Compose Applications of Well Demarcated Parts

P2 can be viewed at different granularities: on the higher granularity, there are the different tiers, which constitute the well-demarcated parts. Within each tier (a lower granularity), there is code, preferable well-demarcated pieces of code, for example components. This is actually how the engineers structure their code; in predefined tiers, and in each tier, different components.

The engineers found that the main gain of structuring applications in tiers is that everyone will work and structure the application in a similar way. This results in coherent, homogeneous and stable applications, leading to easier maintenance. The biggest asset is the standardization benefit, namely that it is predefined where different type of code is located, thus enhancing collaboration (like quickly get into each others' applications).

The statement '*it is easier to program if some logic is put into the display tier*', from one of the interviewees, may imply that full understanding of the benefits of tier separation has not been achieved. It is difficult to say though, because other engineers had a more conscious departure from the strict tier-separation, with clear arguments of why they did what they did. They expressed that they would place input controls, such as checking that correct values are filled out in a form, as well as event handling in the display tier. This reduces transfer over the network and increases performance. Sometimes they would do the input controls twice, both in the display and in the business tier, to have all the logic gathered in the same place, and for extra security. A third-part component demanded some logic to be put in the presentation tier; it was not possible to solve it otherwise. This shows that they have done conscious adjustments of the method prescription to achieve some articulated benefits. If there are drawbacks, and whether the engineers are aware of these is beyond the scope of this investigation. For example, if one starts to add logic into the display-tier, it will not be possible to use a designer without programming experience any longer. Since there is no designer role assigned, this drawback is probably not prominent in their situation.

The learning threshold appeared to be the main drawback with tier-based development. The topic came up a few times in each interview when addressing issues of understanding, updating and extending applications. The engineers experienced the learning process as incremental, and in the beginning it was more difficult to understand systems structured in tiers (a necessity for making updates and extensions). It took about a year to achieve proficiency in extending and updating applications using the separation principle. Most engineers thought it would take about the same amount of time to create a tier-based application as one where display and business logic is entangled. They found it a little difficult to compare though, since they had usually developed smaller systems before. As one engineer expressed it '*larger systems require much more planning and structure, especially if they are being updated later on*'. There seems to be a common opinion that the benefits of tier separated applications mainly appears when building larger systems.

The engineers found both advantages and disadvantages with regards to error handling. If it was obvious in which tier the bug was, the tier structure was advantageous, otherwise you have to run up and down in the tiers, actually taking more time, making it a disadvantage.

The engineers apparently aim to fulfil the goals attached to this method prescription, i.e. there is a foundation for rationality resonance. A closer look at the goals shows that the engineers did aim for G3 (Problems/bugs are easy to find), but did not think tier separation always helps achieving it. Tier separation did help achieving G6 (Applications are easy to update/extend). The main gain was G7 (Applications are built with a standardised coherent architecture).

The underlying values also match: The fact that applications did get extended/updated (V3) may have contributed to the engineers' interest in building general reusable components, and their positive attitude to the ones that had been developed so far. They kept to the tier-separation in most cases, even though it took some extra work. This indicates they do think it is worth the extra effort to prepare an application for extensions/updates (V1) and that they do think in long-term (V2). The engineers were of the opinion that applications composed of well-demarcated parts are easy to extend (scale) and update (V4). The largest application developed in this organization so far has a couple of hundred users. Scaling is therefore not relevant to talk about since the applications are too small. When it comes to locating and determining problems/bugs (V5), tier-based applications had both advantages and disadvantages. The main gain of tier-based architecture was considered the standardization benefit, which underpinned common understanding and collaboration (V13) and that demarcation enhances understanding (V7).

P3: Reuse Components Within and Between Projects

This prescription is very closely coupled with the previous one, so the interview findings from the above section contributes to the understanding of this section too.

The engineers experienced that updating the application often led to partly rebuilding it. At this stage they often realized that things could be done in a more general manner. Through this type of development, general reusable components emerged, creating a library of components within the organization. This was a conscious process led by a project group, with the aim to create their own standard.

The engineers definitely aim to reuse components (G5). They certainly also believed that reusing a component is less work than building a new one from scratch (V11). Developing a general component does take a little more time than developing one for a particular application, but is paid back in the next application. It is clear that reuse of components leads to standardization (V12), which is good for common understanding and for collaboration (V13).

P4: Divide Labour According to Different Skill Sets
The engineers had general competence in the different types of skills needed for developing applications. The demarcations into different skill sets appear on the component level, not on the tier level (as in display-designer versus programmer) as the method-in-concept advocates. Some engineers were slightly more specialized in database programming, while others had a bit more feel for user interface programming. There are no designers without programming experience within the organization, so the benefit of being able to use them to create the display-tier could not be explored.

Goal G4 (team members divide labour between them according to different skill sets) is not really a goal of this organization. The values associated with it do not correspond to the engineers' values either. Since the members of the development team do not have different skills (V9), it is not important to demark what is relevant to each competence (V8) and members of a development team actually can accomplish each other's tasks (V10). This is, of course, specific to this organization.

Value V6 (Applications composed by demarcated parts divides the application into different parts, suiting each skill set) must (just as above) be discussed on the two different granularity levels. On the component level, this is partly true, but on the tier level there is no division into skill sets. The same reasoning applies to V7 (Demarcation enhances understanding).

4.3 Discussion

From the above we can see that the engineers in most cases did conscious departures from the method, which is in line with previous research [6, 25]. The method prescriptions did not make sense in their strict form and were therefore not useful, which is a must for successful method tailoring [2, 22].

The statement '*it is easier to program if some logic is put in the display-tier*' may imply that there is a lack of understanding (as suggested in [5]), and also a sign of short-term thinking [19]. This can be viewed as a goal contradiction – that the personal process goal (easy to program) is favoured before the quality of the product (software); aiming for one goal, gives negative results for another. This issue, to actually put logic into the display-tier is a bit contradictive to the result that the main gain of structuring applications this way is that you know where different type of code is located. The actual rules about what should be put where were appreciated since it made it easier to understand each other's applications. Perhaps the idea about enforcing separation [19] would increase this gain even further? The idea that engineers think in short-term does not apply to the bigger picture of this study though.

The fact that the engineers had to balance different factors and prioritized, e.g. security and performance above keeping to the principle is not surprising. The existence of contradictory goals in ISE is well-known, and the trade-offs between the goals and values brought to the fore in this study and other ISE goals and values would be interesting to explore further. Although this is beyond the scope of this study, the same analytic framework could likely be useful in such an endeavour.

5 Conclusion

Generally speaking, engineers do not keep to the principle of separating business logic from display because in some respects it does not help achieving their goals. In these cases, engineers make conscious departures from the method prescriptions, thus tailoring the method to suit their needs.

This study also revealed more specific reasons for tailoring the principle: Business logic was sometimes placed in the display-tier because '*it made it easier to program*', '*it improved performance*', and '*a third part component demanded it*'. All engineers in the study had similar skill sets, so this caused no misunderstandings. Multiple types of clients hardly ever occurred in the study. The engineers were confident that an application could be efficiently optimized for one type of client; a benefit that is lost in case of systems with multiple types of client. This issue is not mentioned in the literature, thus indicating either that the drawbacks are not explicated, or that the engineers have not understood the method-in-concept. If the drawbacks are suppressed in the method-in-concept, this may be the first and foremost answer to why engineers choose not to follow it.

The principle of separating business logic from display is used in a wide range of software engineering efforts today. It is also well known that engineers are 'cheating' with it, thus potentially deteriorating the quality of the software [19]. This investigation contributes to our understanding of why engineers do this, hence holds the key to how this can be overcome. Non-strict use of the principle gave other benefits, not explicated by the method-in-concept, showing that method tailoring is important. The study also shows that the concept of method rationale is a useful tool for addressing these issues.

This qualitative study provides examples of why developers do not keep to the principle of separating business logic from display – and, in line with previous research, suggests that departure from the principle is often conscious and well-motivated. Given the small scale of the study, hard conclusions are obviously difficult to draw. For more generalizable results, a larger study including several more engineers would be required. It would also be interesting to explore to what extent the same results would appear in a different development environment. Perhaps the results from this study are particular for the .NET environment, whereas other issues could be connected to other development environments. For example, the principle obviously relates to the Model View Controller (MVC) pattern which is widely used (it can indeed be seen as subset of the MVC pattern). In order to understand the influence of contradictions between higher level goals, the interplay between this principle and other software engineering principles needs to be studied as well.

References

1. Ågerfalk, P. J., Åhlgren, K.: Modelling the Rationale of Methods. In: M. Khosrowpour, (ed.): Proceedings of the 10th Information Resources Management Association International Conference. (1999) 184-190
2. Ågerfalk, P. J., Fitzgerald, B.: Methods as Action Knowledge: Exploring the Concept of Method Rationale in Method Construction, Tailoring and Use. In: T. Halpin, J. Krogstie, and K. Siau, (eds.): Proceedings of EMMSAD'05. (2005) 413-426
3. Ågerfalk, P. J., Wistrand, K.: Systems Development Method Rationale: A Conceptual Framework for Analysis. In: Proc. 5th International Conference on Enterprise Information Systems (ICEIS 2003) 185–190
4. Brinkkemper, S., Saeki, M., Harmsen, F.: Meta-Modelling Based Assembly Techniques for Situational Method Engineering. Information Systems 24 (1999) 209-228
5. Dewan, P.: Teaching Inter-Object Design Patterns to Freshmen. In: Proc. SIGCSE'05 (2005)
6. Fitzgerald, B.: The Use of Systems Development Methodologies in Practice: A Field Study. Information Systems Journal 6 (1997) 201-212
7. Fitzgerald, B., Russo, N. L., Stolterman, E.: Information Systems Development: Methods in Action. McGraw-Hill, Berkshire, UK (2002)
8. Forsberg, C., Sjöström, A.: Pocket PC Development in the Enterprise. Addison Wesley, London, UK (2002)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
10. Goldkuhl, G.: Design Theories in Information Systems: A Need for Multi-Grounding. Journal of Information Technology Theory and Application 6 (2004) 59-62
11. Goldkuhl, G., Röstlinger, A.: Joint Elicitation of Problems: An Important Aspect of Change Analysis. In: D. E. Avison, J. E. Kendall, and J. I. DeGross, (eds.): IFIP WG8.2 on Human, Social, and Organizational Aspects of Information Systems Development. (1993) 107–125
12. Jayaratna, N.: Understanding and Evaluating Methodologies. McGraw-Hill, London (1994)
13. Karlsson, F.: Method Configuration, Method and Computerized Tool Support. Doctoral Dissertation. University of Linköping (2005)
14. Levi, N.: Java 2 Web Developer Cerification Study Guide. SYBEX inc., Alameda, California, USA (2003)
15. Lings, B., Lundell, B.: Method-in Action and Method-in-Tool: Some Implications for CASE. In: Proc. 6th International Conference on Enterprise Information Systems (ICEIS 2004) 623-628
16. Nash, M.: Java Frameworks and Components: Accelerate Your Web Application Development. Cambridge University Press, Cambridge, UK (2003)
17. Parnas, D. L.: On the Criteria to Be Used in Decomposing Systems into Modules. Communications of the ACM 15 (1972) 1053-1058
18. Parnas, D. L., Clements, P. C.: A Rational Design Process: How and Why to Fake It. IEEE Transactions on Software Engineering 12 (1986) 251-257
19. Parr, T.: Enforcing Strict Model-View Separation in Template Engines. In: Proc. WWW2004, ACM (2004)
20. Patton, M. Q.: Qualitative Evaluation and Research Methods. SAGE Publications, Newbury Park California, USA (1990)

21. Pree, W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley (1995)
22. Riemenschneider, C. K., Hardgrave, B. C., Davis, F. D.: Explaining Software Development Acceptance of Methodologies: A Comparison of Five Theoretical Models. IEEE Transactions on Software Engineering 28 (2002) 1135-1145
23. Rogue Wave Software, I.: Distributed MVC: An Architecture for Windows DNA Applications. Boulder, Colorado USA (1999)
24. Rossi, M., Ramesh, B., Lyytinen, K., Tolvanen, J.-P.: Managing Evolutionary Method Engineering by Method Rationale. Journal of the Association for Information Systems 5 (2004) 356-391
25. Russo, N. L., Stolterman, E.: Exploring the Assumptions Underlying Information Systems Methodologies. Information Technology & People 13 (2000) 313-327
26. Stolterman, E., Russo, N. L.: The Paradox of Information Systems Methods: Public and Private Rationality. In: Proc. The British Computer Society 5th Annual Conference on Methodologies (1997)
27. Sun Microsystems, I. Java Blueprints: Model-View-Controller. Sun Microsystems, Inc [Online]. Available: http://java.sun.com/blueprints/patterns/MVC-detailed.html
28. Yin, R. K.: Case Study Research, Design and Methods, 2nd Edition. SAGE, London (1994)