A Case Study of Horizontal Reuse in a Project-Driven Organisation

Henrik Bærbak Christensen and Henrik Røn
University of Aarhus
Centre for Experimental Computer Science
DK–8200 Aarhus N, Denmark
Tel.: +45 8942 3188
E-mail: {hbc,roen}@daimi.au.dk

Abstract

This experience paper presents observations, lessons learned, and recommendations based on a case study of reuse. The case study is concerned with the development, maturation, and reuse of a business domain independent software component (horizontal reuse) in a project-driven organisation that has little previous experience with systematic software reuse. The main lessons learned are that: (a) even though domain analysis can alleviate reuse mismatch problems one should not underestimate the technical problems that may arise when reusing; (b) a side-effect of reuse is that software engineering knowledge is transferred within an organisation; (c) design patterns can be as risky as they can be beneficial; and (d) there is more to architectural mismatch than "merely" packaging mismatch.

1. Introduction

Software reuse is widely accepted as one of the key techniques to improve productivity and quality in software engineering. Time has shown, however, that this apparently simple idea is elusive and difficult to master in practice. The reuse community has produced a number of excellent books [2, 14, 16, 26] but from practitioners' point of view, some of the focus and recommendations are difficult to apply in a concrete context. We attribute this to a number of factors:

- The predominant form of reuse covered in the literature is product-line reuse. Prerequisites for product-line reuse include a well-known business domain and a large customer potential. However project-driven organisations, which often do not have core business domains, can not use a product-line strategy and must therefore rely on horizontal reuse. Unfortunately literature discussing horizontal reuse is scarce.
- Many recommendations are rather abstract and difficult to evaluate within a given development context.
- Though textbooks and the folklore stress an incremental introduction, there is little information to be found about "what are the things to do first".

Griss argues for the need for community consensus on the "reuse body of knowledge" [12]. We agree—even though many authors present various techniques, maxims, and rules of thumb they are presented in an incoherent way which makes them difficult to compare and analyse for a practitioner.

Our case study is concerned with horizontal reuse, that is, reuse across business domains. Horizontal reuse is intrinsically difficult: as there is no business domain given a-priori, there is no overall software architecture to define the relations and interfaces between the reusable assets. This often (or rather usually) leads to *architectural mismatch* between assets that makes it anything from tedious to very difficult to make them work together [11, 21].

The purpose of this paper is to report experiences and recommendations which cover aspects of the horizontal reuse region of the reuse space. It is our hope that the information provided can be helpful in two ways. First, it may be helpful for teams and companies in a situation similar to ours. Second, we hope our data can be useful as input for a catalogue of reuse best practices.

2. Background

Our case study is concerned with the development and reuse of a software asset. The asset is a graphical editor for constructing and editing search expressions over an arbitrary data domain. As an example, it is possible to instantiate it to handle the domain of e-mail messages; this would allow a user to construct a search expression like *(Subject='COT' OR Subject='CIT') AND Receiver='hbc' AND Date after '01-01-2000'*. The asset is implemented in C++ as an object-oriented white-box framework and will be briefly outlined later. We will use the term "asset" and

“framework” interchangeably to denote the reusable software entity.

The primary purpose of the pilot case study was to get first hand experience with fundamental reuse issues in a concrete industrial context [25]. The limited setting of the pilot enabled us to disregard organisational- and people-issues which are usually considered more important for success on a company-wide scale [14]. A secondary influence on the case study was an actual need for the asset, namely two distinct projects that needed to implement similar functionality. This need was identified informally.

The first project, IRIS/MFS, is responsible for the continued development of a mature product for handling structured electronic messages, i.e., a highly specialised e-mail tool. The second project, EWare, develops a system for handling and filtering large databases of radar signatures. In IRIS/MFS searching is used to find particular structured messages; in EWare searching is used to define subsets of a database through filtering.

The technical setting for the case study was that both projects used C++ but used different compilers and development environments (here understood as conventions for organising source files, “make” program, etc.).

3. The process

The framework was first developed in the IRIS/MFS project and then later reused in the EWare project. The development and deployment process is outlined in this section. An overview of the design can be found in [6], while a detailed account can be found in [19, 7].

3.1. Design process

A number of analysis and design meetings produced an initial overall design for the asset. Developers from the IRIS/MFS (asset producer project), EWare (asset reuser project) and researchers from university participated in these meetings. This approach takes the recommendation of the reuse domain analysis community: to consider the requirements of potential future reusers, one step further as the reusers were directly involved in the design. This additional cost for the EWare project amounted to about 3–4 man-days.

The challenge was to construct a reusable framework that could to be operational within both projects. A closer analysis revealed four major areas where the framework had to be decoupled:

1. *Domain data*. The two application domains have very different data models. The electronic mail domain has only one entity type, message, with a fixed number of attributes. The radar signature domain has an unlimited number of entity types and the number of attributes of each entity may vary from customer to customer. Thus, the client of the framework must be able to specify the entities and attributes of the domain data.
2. *Expression execution*. A consequence of the need to handle arbitrary domain data is that the asset can not assume anything about how the domain data is searched. Still, it must produce an expression that facilitates the actual execution of the search by the client. Thus, the asset must be decoupled from the actual search algorithm.
3. *Representation of expression*. A consequence of the fact that the search expression must be executed by the client is that search expressions must be passed between the instantiated framework and the client. Thus, the representation of the search expression must be independent of the business domain.
4. *Graphical user interface*. The search expression is edited by a user through a traditional graphical user interface dialog. The project applications run on both UNIX and Windows NT. Thus, the asset must be decoupled from the specific graphical user interface toolkit.

The resulting design relies heavily on design patterns to achieve the decoupling. The *Reflection* [4, p. 193] pattern is used to decouple the entities and attributes of a specific data domain from the framework by defining a meta-model of the domain specific entities and attributes. The meta-model is given as a parameter to the framework when it is instantiated. The search expression is defined by a *Composite* [10, p. 163] pattern where leaf nodes define values of atomic attributes indirectly through references to the meta-model entities, e.g., a leaf node could be Subject=‘COT’. The framework defines an abstract *Visitor* [10, p. 331] pattern for traversing a search expression and the client must subclass this to perform the actual execution of the searching in the domain data of the client. The execution is thus the responsibility of the client but the framework defines a template for the source code structure. Finally, the *Adapter* [10, p. 139] pattern is used to define an abstract, platform-independent, graphical user interface toolkit, and the framework provides backends for both X11 and Windows.

The graphical user interface of the framework is shown in Fig. 1. Pane ① is a list-box listing all attributes of the client application’s domain. The framework queries the meta-model to produce the list. An attribute editor, pane ②, is available for each type of attribute. The framework supplies a number of types such as string, number and time, along with corresponding attribute editors. The composed search expression, pane ③, is pretty printed using a supplied instance of the visitor patterns; the client may supply its own for complete formatting control. The buttons, pane ④, allow individual atomic attribute expressions (from pane ②) to be added using the logical operators AND, OR, and NOT, or removed from the composed expression.

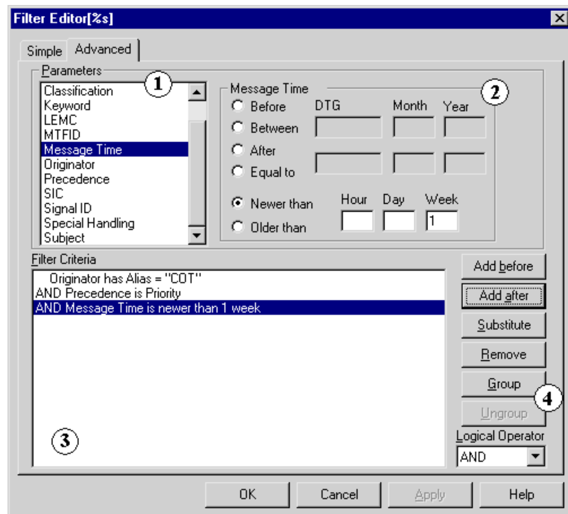

Figure 1. The expression editor.

3.2. First development

The framework was developed in the IRIS/MFS project. An IRIS/MFS developer who had participated in the design meetings was appointed as lead framework developer. It was the plan that he should implement it alone, but due to a rescheduling of the project to meet an earlier delivery date, the implementation task was staffed with four additional developers. None of these developers had much experience with design patterns and due to the time pressure no coordinated training was carried out. In this changed context the use of design patterns became a serious problem. The overall design, heavily relying on patterns, was difficult to understand for the new developers. This caused frustration among some of the new developers. The implementation of a design pattern must obey a lot of rules in order not to break the intent (and thus benefit) of the pattern. For the newcomers it was difficult to judge when their coding effort actually followed these rules or inadvertently broke them. Furthermore patterns are not blueprints for solution as one must making a number of important decisions regarding the actual implementation of the pattern. It proved difficult for the additional developers to make these decisions and some of their decisions inadvertently reduced the reusability of the framework.

3.3. Harvesting and maturing

During the harvesting and maturing process the framework was extracted from the IRIS/MFS project, made operational in a isolated test environment, generalised and documented. Due to the inexperience of the added developers some of the code, though working, did not follow the intent of the underlying design and compromised the intended decoupling. This required quite a bit of code rewriting.

The main documentation consisted of a number of implementation scenarios and a small demonstration program showing how to use the framework. The implementation scenarios were based on the concepts of *pattern language* by Johnson [15] and *motifs* by Lajoie and Keller [17]. The documentation consists of a description of when and how the framework is applicable, an overview of the concepts in the framework and finally references to a number of implementation scenarios describing in detail (a) how to solve certain tasks using the framework, and (b) how to customise the framework.

3.4. EWare reuse

The framework was handed over to the EWare project for reuse. The framework was introduced to the EWare developer in a design walkthrough meeting followed by the lead framework developer's opinion on how to best customise the framework to satisfy the requirements of the EWare customer. After the meeting the EWare developer wrote a design document that was reviewed by the lead framework developer. The implementation process embodied: (a) getting the framework up and running in the EWare software context; (b) defining the meta-model; and (c) implementing the visitor that transforms the search expression into SQL for the underlying relational database. The implementation effort was tracked quite closely as reported in the next section.

Point (a) turned out to be costly. Much effort had to be invested just to make the framework compile, link, and run using the EWare compiler, make program, and environment. Many of these problems required changes in both the underlying framework code as well as non-code artifacts such as makefiles, graphical resource files, physical file organisation, etc. These changes were costly as they of course had to be retrofitted into the original IRIS/MFS project to avoid a multiple version maintenance problem.

The EWare developer's overall impression of the process was favourable. The functionality provided by the framework clearly met the requirements of the customer, provided a sound architecture for solving the problem at hand and the EWare developer himself had learned (some) design patterns thoroughly—knowledge that he considered valuable in future projects. However, he felt that the framework was heavy-weight in the sense that many issues had to be understood and implemented before it could produce meaningful functionality: Defining meta-model, attribute editors, visitors, etc.

4. Analysis

The data from the case study is primarily qualitative sources:

- *Development diary:* Both the lead framework developer and the EWare developer kept a diary throughout the

Table 1. Problem sources and costs.

Source/Cost	Low	Medium	High	*Sum*
Technical	7	1	2	10
Design	5	-	-	5
Process	-	1	-	1
Training	3	-	-	3

process. The diaries informally describe their problems and experiences.

- *Logged staff hour:* All spent staff hours were registered in a central system.
- *Evaluation interviews:* About once a week during the reuse process the EWare developer was interviewed about encountered problems and progress by a group of university researchers. The interview also involved discussion of technical issues and their solution.

Based on these sources all problems encountered in the various processes were listed and classified according to two dimensions (possible values are given in square brackets):

- *Problem source [Design, Technical, Process, Training]:* A classification of the underlying source of the problem. The categories are described below.
- *Cost [Low, Medium, High]:* The coarse-grained cost of finding a suitable solution to the problem, measured as the relative amount of staff-hours spent to correct the problem.

The problem source categories are defined as follows:

- *Design* classifies a problem as a design flaw i.e. the framework was unable to provide the functionality required by the EWare customer.
- *Technical* classifies a problem as a technical problem i.e. the framework could not compile or link in the EWare development environment.
- *Process* classifies a problem as a process-related problem, caused by time-pressure in the framework development. The sole problem classified as such was the lack of proper documentation of the framework API.
- *Training* classifies a problem as due to the inexperience of the EWare developer with techniques and tools used in the framework.

The reuse process encountered 19 problems that had to be corrected. Table 1 shows our classification of problems with respect to the source of the problem and the cost (details are given in [6, 7]).

Our main conclusion from this problem classification is that our initial analysis and design effort paid off as only a few, low cost, design problems were encountered. However, we had underestimated the costs of technical problems. Some of the major technical problems were:

- The asset depends on the C++ template container library STL. The EWare project compiler did not provide STL. A public domain STL suitable for the compiler was found. It turned out, however, that the compiler could not compile the STL library. A compiler patch was found at the compiler vendor's web-site. This was a high cost problem and potentially critical if a suitable STL library and compiler patch had not been found.
- The EWare development environment ("make" program, conventions on physical file locations and resource file layout, etc.) was radically different from the IRIS/MFS environment and much time was spent on adapting the framework's makefiles etc. to the EWare environment. This was a high cost problem.
- The asset contained a C++ cast to a subclass in a multiple fork-join inheritance hierarchy that is illegal according to the C++ specification [23], but the IRIS/MFS project compiler did not report the error and produced "working" code. The EWare project compiler, however, did not accept the cast. This was a medium cost problem.

We acknowledge that our classification of problem sources is not completely unambiguous. However, as the three problems outlined are definitely technical problems and were the most costly to correct, we find that any ambiguity in the classification of the remaining problems can not invalidate our main conclusion.

4.1. Reuse metrics

The EWare project originally made an estimate of 260 staff-hours to implement a search facility from scratch. As the corresponding estimate made in the IRIS/MFS was considerably higher this number may be a bit optimistic.

The actual effort invested by EWare in reusing the framework amounts to 199 staff-hours spent on training, designing, implementing, and testing a solution using the framework, and finally documenting the solution. Additional 79 staff-hours were spent on the outlined technical problems, mainly by the lead framework developer. When one considers the additional 263 staff-hours invested by IRIS/MFS on extracting, maturing, and documenting the framework, it supports accepted wisdom that a reusable asset can not become profitable before it has been reused at least three times. It also underlines that technical problems may be relatively costly even though some of the outlined problems will not reappear in future instances of reuse.

5. Lessons learned

In this section we discuss the analysis above as well as the more qualitative observations made in the case study.

The reuse literature emphasises commonality- and variability analysis with the given business domain. Our case

study supports this observation also within "lateral domains" [22] understood as partial but well-defined functionality that is relevant across business domains. The involvement of potential reuse clients directly in the design phase was recommendable. We find that face-to-face communication is important and that the design profitted from the experiences of the EWare project. The strength of the approach is also suggested by the fact that the initial overall design remained stable throughout all processes.

The emphasis on domain analysis, however, lead us into the pitfall of overlooking the technical issues. This proved to be costly.

A code asset embodies more than just the source code and mismatches between assets can stem from other than syntactic and semantic sources at the source code level. In our case, much effort was put into adapting the makefiles and user interface resource files, a process somewhat similar to the "recycling of non-code artifacts" reported by Allen et. al. [1]. Instances of such *technical mismatch* may severely lower reusability as required, non-code, artifacts have to be adapted for every (re)use—essentially depriving the asset of being a black-box asset. One key obstacle is the "make" program itself [9], which is inherently non-modular. The reuse community has proposed modular replacements for make (e.g. Object Make [24]) but if these will succeed remains to be seen; the software configuration management community has presented a large number of better build programs for more than a decade with little success.

Technical mismatch problems seem to have received little attention compared to packaging mismatch problems. In the paper by Garlan et al. [11] the term "architectural mismatch" included problems with incompatible build scripts, but later papers focus on the problems of mismatch in a more narrow sense as differences in interfaces, data formats, push/pull models of control flow, etc. [3, 8, 21]. The standard "solution" for technical mismatch problems is that of *avoidance*: "Standardise tools and environments". However there is a limit to standardisation as various stake-holder influences such as customer requirements, legacy systems, multi-platform support, etc., may oppose a standardisation effort and then only the discipline of the developers can lower technical mismatch problems through following rigid coding conventions, style guides, etc. Binary component technology may be seen as a solution, but at present the competing technologies themselves present a technical mismatch problem.

The pilot case study had several spin-off benefits. The most important was transferring software engineering knowledge in the organisation. The reuse developer had no previous experience with design patterns but the initial design walkthrough meeting followed by the concrete work of reusing the search framework made him become intimately acquainted with a small set of them. To many developers applying a new technique and being rewarded immediately by a working program is a more fun way of being introduced to a new subject than an abstract introduction such as reading a book. The reuse developer reported design pattern knowledge as his main benefit of the exercise.

Design patterns proved to be beneficial for communicating a design and, of course, they acted as a repository of well-proven solutions to many of the problems faced. However, they also presented themselves as a risk if not accompanied by proper training. As described in Section 3 the expansion of the team during the initial implementation phase exposed a potential cost of using advanced techniques: The new team members had no experience with design patterns and did not receive properly coordinated training. Especially one of them had great difficulties in understanding them and even more in producing code that followed the intent of the design and ensured the desired decoupling. This was costly because of subsequent code rewrite, and caused frustration within the team.

Mili points out that generality and immediacy are contravariant properties of assets [18]; making an asset more general makes it less immediate to use. Our case study supports this observation. The search asset is general in the respect that it can be extended with new datatypes and editors for these. This, however, calls for a rather complex design and as a consequence the reuse developer must follow a number of guidelines and steps to customise the asset properly. The reuse developer found these steps easy to follow but laborious—it took a lot of effort to get the first working program. Making an asset too general can make it "reuseless" as the investment required to understand and use an asset may become prohibitive.

Finally, a reuse pilot project can highlight the importance of other improvement initiatives in the organisation. An ongoing discussion is to standardise the range of tools and environments as much as possible under the aforementioned constraints; our case study has underlined the importance of this initiative.

6. Summary

We have tried to condense our experience into four recommendations:

- *Do not underestimate technical issues.* The emphasis in literature on organisational- and process issues, domain analysis and product line reuse should not make one forget the technical issues. If the technical environment does not facilitate reuse, reuse can not happen, and consequently these issues must be addressed at an early stage of a reuse initiative. Standardisation of tools, environments, and enforcement of coding- and style conventions are the present answers to technical mismatch problems.

- *Advanced software engineering techniques is both an reuse enabling technology as well as a risk.* Design patterns are central to our solution's success in decoupling our asset from the concrete domain data. However, a pattern oriented design is difficult to communicate to developers with little design pattern experience.
- *Technical mismatch may inhibit black-box reuse.* An asset is more than just the source code. It also embodies the build scripts, user interface resource files, graphical style, etc. If any of these parts require manual adaption, the asset cannot be considered a black-box asset. At the source code level design patterns provide feasible decoupling solutions; similar decoupling techniques for non-code artifacts are not widely accepted nor accessible.
- *Reuse may be an excellent way to transfer software engineering knowledge in an organisation.* Many developers are more inclined to learn new software engineering techniques when they see an immediate, productive, benefit. We also think that they learn difficult techniques (like the intrinsically complex visitor pattern) better by seeing how they are used rather than by reading a book or attending a course.

We regrettably found literature discussing "horizontal reuse" from a practical point of view scarce. We therefore find it an important issue to compile a catalogue of "reuse best practices". If possible, we would like such knowledge codified into "reuse patterns" that will allow practitioners and organisations to assess the applicability of the various techniques as well as define a common vocabulary of strategies.

Acknowledgements

This research has been funded by the *Centre for Object Technology* [5]. The Centre is financially supported by The Danish National Centre for IT-Research (CIT) and the Danish Ministry of Industry.

References

[1] J. Allen and L. Latour. If you Can't Reuse, Recycle: A Case Study of a Platform to Platform Port. In IWSR8 [13]. URL: http://www.umcs.maine.edu/~ftp/wisr/wisr8/wisr8.html.

[2] P. G. Bassett. *Framing Software Reuse – Lessons from the Real World*. Yourdon Press, Prentice Hall, 1997.

[3] F. Bronsard, D. Bryan, W. V. Kozaczynski, E. S. Liongosari, J. Q. Ning, A. Olafsson, and J. W. Wetterstrand. Towards software plug-and-play. In *Proceedings of the 1997 Symposium on software reusability SSR'97*, pages 19–29, Boston, United States, May 1997.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture*. John Wiley and Sons, 1996.

[5] Centre for Object Technology. *COT home page*. http://www.cit.dk/COT.

[6] H. B. Christensen and H. Røn. A Case Study of Framework Design for Horizontal Reuse. In *Proceedings of 37th International Conference on Technology of Object-Oriented Languages and Systems, TOOLS Pacific 2000*. IEEE Computer Society Press, 2000. To appear.

[7] H. B. Christensen and H. Røn. Case Study of Maturing and Reusing a Framework. Technical Report 3-32, Centre for Object Technology, April 2000. URL: http://www.cit.dk/cot.

[8] R. DeLine. A Catalog of Techniques for Resolving Packaging Mismatch. In *Proceedings of the Fifth Symposium on Software Reusability*, pages 44–53, Los Angeles, CA, USA, May 1999. ACM SIGSOFT, ACM Press.

[9] S. I. Feldman. Make – A Program for Maintaining Computer Programs. *Software – Practice and Experience*, Apr. 1979.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reuseable Object-Oriented Software*. Addison-Wesley, 1994.

[11] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, Seattle, Washington, April 1995.

[12] M. L. Griss. Reuse 2001-2004: What next, now that we have solved all reuse problems? In *Ninth Annual Workshop on Institutionalizing Software Reuse*, 1999. URL: http://www.umcs.maine.edu/~ftp/wisr/wisr.html.

[13] *Eighth Annual Workshop on Institutionalizing Software Reuse*, 1997. URL: http://www.umcs.maine.edu/~ftp/wisr/wisr8/wisr8.html.

[14] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture Process and Organization for Business Success*. ACM Press, 1997.

[15] R. Johnson. Documenting frameworks using patterns. In *OOPSLA 92*, pages 63–76. ACM, 1992.

[16] E.-A. Karlsson. *Software Reuse – A Holistic Approach*. John Wiley and Sons, 1995.

[17] R. Lajoie and R. K. Keller. Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert. In *Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l'Avancement des Sciences (ACFAS)*, Montreal, Canada, May 1994. Colloquium on Object Orientation in Databases and Software Engineering.

[18] A. Mili, S. Yacoub, E. Addy, and H. Mili. Toward an Engineering Discipline of Software Reuse. *IEEE Software*, 1999.

[19] H. Røn. Development of a Component for Searching and Filtering. Technical Report 3-20, Centre for Object Technology, May 1999. URL: http://www.cit.dk/cot.

[20] M. Samadzadeh and M. Zand, editors. *Proceedings of the 1995 Symposium on software reusability SSR'95*, Software Engineering Notes, Seattle, Washington, Aug 1995. ACM, ACM Press.

[21] M. Shaw. Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging. In Samadzadeh and Zand [20], pages 3–6.

[22] M. A. Simos. Lateral Domains: Beyond Product-Line Thinking. In IWSR8 [13]. URL: http://www.umcs.maine.edu/~ftp/wisr/wisr8/wisr8.html.

[23] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1997.

[24] Y. Sugiyama. Object Make: A Tool for Constructing Software Systems of Existing Software Components. In Samadzadeh and Zand [20], pages 128–136.

[25] Systematic Software Engineering A/S. *Additional product information on IRIS/MFS and EWare*. http://www.systematic.dk/products/products.html.

[26] W. Tracz. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Addison Wesley Publishing Company, 1995.