In C. Paris, N. Ozkan, S. Howard, and S. Lu (eds.), *OZCHI 2000 Conference Proceedings: Interfacing Reality in the New Millennium* (Sydney, Australia, December 4-8, 2000), pp. 88-94. CSIRO, North Ryde, Australia.

Preprint version

Component-Based Design May Degrade System Usability: Consequences of Software Reuse

Morten Hertzum
Centre for Human-Machine Interaction
Risø National Laboratory, Denmark
morten.hertzum@risoe.dk

Abstract

Component-based design is gaining attention as a potentially feasible approach to software reuse. An important aspect of this reuse potential lies in the possibility of turning existing applications into functionally rich, reusable components and, thereby, preserving the investment in legacy systems. Based on fieldwork in a software development company where this practice has been adopted, this study analyzes how the consequences of component-based design reach beyond the development process and well into system use. It is argued that functionally rich components add new complexities to the mapping between the system and the real world, and may lead to degraded system usability. In the field study, the potential usability issues involved in relying on functionally rich components include a fragmented system image, task gaps, conceptual mismatches, rekeying, scalability problems, and added education and training. Systems development companies should be wary not to uncritically adopt techniques that support reusability at the expense of usability.

Keywords : Component-based design, reuse, usability.

1. Introduction

Software reuse has long been expected to result in substantial productivity and quality gains but to date this expectation has been largely unmet [2, 3, 12]. On the surface, software reuse seems the most obvious of software practices but the few success stories and many reuse failures evidence that it is truly difficult to put software reuse into actual practice. When systems developers create software, they make extensive use of knowledge they already possess. This type of reuse of one's own experiences, possibly in the form of code fragments, is the essence of professionals' ability to gain proficiency through experience [4]. The difficulties arise when reuse is attempted in cooperative settings where multiple systems developers are involved in making, maintaining, using, and possibly reusing design ideas and code fragments over extended periods of time.

Table 1 lists some of the major reasons for reuse failures, which have been identified in the literature. It is apparent that these reasons concern the context in which the software is developed. This study investigates the consequences of software reuse on the *usability* of a system where reuse is approached through component-based design. In the studied systems development company, the recent adoption of component-based design is considered necessary to the efficient development of high-quality software. However, a field study of one development project within the company reveals a recurring discussion among the project participants regarding the viability of building the system from functionally rich, reusable components as prescribed in the project plan. While the development-side benefits of component-based design are undisputed,

Table 1. Reasons for reuse failures [2, 10, 12, 13]

1.	Lack of support and long-term commitment from management
2.	Corporate culture and reward system discourage reuse
3.	Belief that reuse is counter-creative
4.	Lack of understanding about why to practice reuse (not-invented-here syndrome)
5.	No experience in practicing reuse and nothing to reuse, i.e. no software reuse library
6.	A belief that the current application is unique and, thus, cannot benefit from reuse
7.	No tools and methodology support
8.	The documentation of the software is either non-existent or insufficient
9.	Software that can perform the required task is available, but it is so general that it is too inefficient for the task
10.	The software performs a task that resembles the required task, but the cost of changing the software to perform the required task is greater than the cost of writing new software

several project participants are severely concerned that building the system from such components will make it prohibitively difficult to use. This study lays out and analyzes these concerns to improve our understanding of how the implications of software reuse reach beyond the software development process and well into systems use. It should be noted that the system is still under construction and, thus, no data on real use are available.

The next section briefly introduces component-based design. Section 3 accounts for the method used in performing the field study, and Section 4 introduces the project that has been studied. Sections 5 through 7 analyze how component-based design may degrade system usability in ways that are only gradually recognized by the studied systems developers. Finally, Section 8 concludes the paper by underlining that the reuse payoff that development organizations expect of component-based design may come at a considerable price.

2. Component-based design (CBD)

Software components are gaining a great deal of attention within systems development as a relaxed approach to object-oriented design and development. Components, as is true with objects, use encapsulation to separate component specification and invocation from component implementation. Thus, components provide an external interface to their functionality and hide all details about the internal constructs that go into providing this functionality. This means that: (1) All communication with a component is through its interface, which is the only thing you need to know about to use the component. (2) As long as the interface remains unchanged, the implementation of a component can be changed without knowledge of where and how the component is used. Contrary to objects, components tend not to embrace the more complex characteristics of object-oriented design and development, i.e. polymorphism and inheritance. This means that there is no component hierarchy where changes to one component produce changes in all the components derived from it. Seemingly, an increasing number of companies find that component-based design may potentially be a simpler, more controllable, and more feasible process [8].

Encapsulation is one important property of components; another is the functionality the components provide. Components are, generally, defined at a level where they can readily be related to specific business processes and are, hence, typically richer in functionality than objects. Sprott [14] finds that adopters of component-based design often concentrate on functionality and place less emphasis on the benefits of encapsulation. Specifically, component-based design seems to progress toward [8]:

- The implementation of more sophisticated business functions.
- Suites of configurable components that can be used as building blocks in developing domain-specific applications.
- Server-side component 'wrappers' for legacy applications and data.
- Turning entire applications into components to achieve modularity and easy interoperability with other applications.

As can be seen from this list an important part of the reuse potential of component-based design lies in the possibility of turning existing applications into components, which can then be reused for additional purposes. This is attractive from a resource perspective because it preserves the investment in the existing applications and provides a way to gradually migrate legacy systems to a client/server or Web environment. For users, the most visible advantage of component-based design is increased consistency in that all occurrences of a specific task are supported with the same component; thus, the user is relieved from random variations in interface or functionality.

3. Method

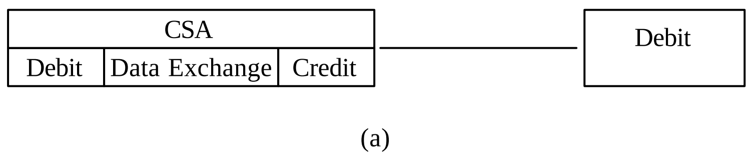
The data collected for this study cover the first eleven months of a two-year system development project. I have followed the project by (1) participating in the two-day start-up seminar, (2) being present at the fortnightly status meetings and some additional meetings, (3) conducting interviews with eleven of the core project participants, and (4) inspecting various project documents. The meetings and interviews have been recorded on tape and transcribed. This study is based on an analysis of the 22 meetings I have observed, supplemented with data from the interviews.

The main purpose of the meetings has been to provide a forum for sharing information about the status of the project, maintaining awareness of the entire project, coordinating activities, discussing problems and progress, making decisions, and reviewing major project documents. During the meetings, I have been seated at the meeting table with the other people present. From their point of view, I have been invisible in that I was not to be spoken to and have myself remained silent. During the breaks, I have talked informally with people.

The interviews provided an opportunity to talk about people's individual experiences and concerns, and to dig deeper into issues and discussions that were merely hinted at during the meetings. The interviews, which lasted 1-1½ hours each, concerned the project participants' roles and responsibilities in the project as well as their views on what was critical to successful completion of the project.

4. The CSA Project

The company where the field study took place is a large software house, which has developed and marketed a range of systems for use in local government



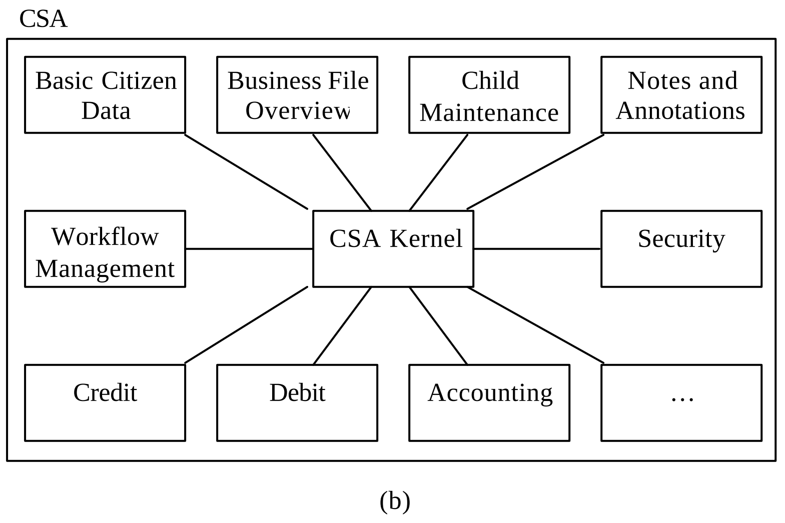


Figure 1. Architecture of the CSA system. (a) The existing CSA system is interfaced to one component developed outside the project. (b) The new CSA system will consist of a kernel and a dozen externally developed components.

institutions. The studied project concerns a system to support local government authorities in the handling of cases concerning child support and alimony (CSA). The CSA project was initiated in 1999 and will, according to the project plan, last two years. The first eleven months of the project, the period analyzed in this study, concerned the requirements specification, the business modeling, and part of the application and component design. During this period, the project was staffed with a project manager, eleven designers/developers, two service consultants, a methods & tools consultant, a usability specialist, and a secretary. The project manager and six of the designers/developers worked full time on the CSA project, the remaining ten persons were assigned to the CSA project on a part-time basis. When referred to as a group, the members of the CSA project will be termed CSA engineers, irrespective of their different educational backgrounds.

The CSA engineers are to completely redevelop the existing CSA system, which several of them have been heavily involved in developing and maintaining over the last 18 years. Whereas the existing CSA system contains substantial amounts of code that duplicate functionality from other systems made by the company, the new CSA system will distribute this functionality onto components that are to be developed by other project groups in the company. The adoption of component-based design means that the CSA engineers have to cooperate closely with a number of people outside the project to negotiate, settle, and follow up on component definitions and how the development of the components progresses [7]. Naturally, the CSA engineers also have to interact with a number of other stakeholders in the development process, including user representatives.

5. CBD from a user-centered point of view

The existing CSA system consists of three modules (Figure 1a). While the data exchange module is specific to the CSA system, many of the systems developed by the company have debit and credit modules. From a bird's eye perspective the functionality of these modules is similar across systems but in each case the modules have been developed based on an analysis of the specific circumstances that characterize this particular use situation. In the case of the CSA system, the debit module provides a carefully tailored CSA-view into the company's standalone debit system. The rationale for this is twofold: (1) The standalone debit system lacks some facilities needed to handle CSA cases. (2) CSA users are working with either the credit side of the system or the debit side. The debit module of the CSA system gives the credit-side users all the debit information they need and, thus, relieves them from the complexities of the standalone debit system. The debit-side users deal with the debit aspects of many kinds of cases besides CSA cases, and therefore need a system that is not tailored to CSA needs only.

The new CSA system will to a large extent be composed of components developed outside the CSA project (Figure 1b). Most of these components are business components brought about by defining interfaces that turn standalone, legacy systems into components. By and large, this reduces the amount of code that has to be produced in the CSA project to the CSA kernel. This reduction is achieved at the cost of a substantial amount of work coordinating and following up on the development of the components [see 5, 7]. However, the adoption of component-based design probably means that the new CSA system will encompass more special cases and support more aspects of the users' work than if the CSA engineers were developing everything themselves. The users – especially the credit-side users who are the primary users of the CSA system – may however experience that the system has become less transparent. Specifically, the new system is not contained within a closed set of screens and functions but will extend into components that (1) are themselves entire systems, (2) do not necessarily comply completely with CSA conventions, and (3) provide other facilities besides those relevant to the handling of CSA cases. This suggests that it will be very difficult to provide the user with a strong and consistent system image [11]. The user is, instead, likely to experience difficulties in forming a coherent conception of how CSA cases are modeled in the system.

The decision to use components to the greatest extent possible was stated in the founding project documents. Whereas the designers/developers initially tended to consider it a purely technical decision, the service consultants were concerned that the decision to develop the CSA system from functionally rich components would also have consequences for the users. As the project progressed, the potential usability issues involved in relying on functionally rich components were gradually realized and brought up for discussion at several meetings. These issues include:

- *Task gap*. When a lot of the system functionality is provided by components developed with other use situations in mind, the coupling between the system and the users' task suffers. It becomes more difficult to see through the system and maintain a focus on the actual CSA work. Instead, it becomes more likely that the user will have to spend time working out the real-world meaning and consequences of various system options and facilities.
- *Conceptual mismatches*. The same concept may be used in several components but it may not mean the exact same thing. For example, a person's income is calculated in different ways in different situations, and some components have codes – such as retirement codes – that are used differently in different components. Often, it takes considerable insight into two components to tell whether a common code actually means the same in both of them.
- *Rekeying*. As the components are rather self-contained, the user will at times be required to key in the same piece of information several times – in different components. This is a trade-off between the development effort required to integrate the components and the manual procedures the users must perform to bridge gaps between the components.
- *Scalability*. The components were originally developed for contexts with a certain load in terms of cases, events etc. Reusing a component in a context with a substantially lower or higher load causes the design to be awkward or inadequate, although the functionality may in principle be right.
- *Education and training*. When tailor-made, local modules are replaced with more versatile and much bigger components, the user experiences fewer restrictions but has to spend more time learning how to use the system.

The above issues are tied to the use of *functionally rich* components. In the CSA project the use of functionally rich components is a result of the decision to turn existing applications into reusable components. While component-based design can certainly be approached in other ways, components are typically at the level of business processes, and the trend seems to be toward increasingly complex components (see Section 2). Biggerstaff & Richter [2] hold that "as a component grows in size, the payoff involved in reusing that component increases more than linearly." Further, opting for small components will normally mean that many more components are necessary and this in turn means that it becomes a task in itself to get to know when an appropriate component is available and which one to choose. Thus, there is reason to believe that components will often be functionally rich and, consequently, that the task gap, conceptual mismatches etc. are usability issues of potential relevance to much component-based design.

6. CBD and the users' key competence

The users of the CSA system are subject specialists characterized by putting to work their intellectual skill learned in systematic education and through experience. They are to a large extent paid to make sense of things and pass judgements, and they do that largely by resort to structures internal to themselves rather than by resort to external rules or procedures. Though the handling of CSA cases is prescribed in detail in written legislation, there is a large gap between the terse texts and the richness of real-world cases. To close this gap, the users of the CSA system have to interpret the legislation with respect to the concrete cases they are confronted with. Over time, this leads to a practice that reflects the legal norms laid out in the legislation but is not inherent in the written legislation as such [9]. One of the service consultants provided an illustrative example of the kinds of cases users consult the call center about:

> Service consultant: We have just had a case where the county has decided that a person should be billed although there is no document [a CSA case is defined by a document that settles who will be paying whom and how much; legally the document is a contract]. There is just an agreement among the parties. It has been discussed in the county twice, and the county has decided that they [the local authorities] shall send out a bill. There is no document. There is an agreement among the parties but that is not a document in any legal sense. They will bill on the basis of it anyway. […] I'm saying this to illustrate – it's just a small selection of one day's calls – what it is they call and ask us about. To illustrate that the rule-basedness you expect is not what we experience.

It is essential to note that using the CSA system does not simply consist in feeding it with input and then accepting the output from the system as the correct decision. Rather, the user first arrives at the correct decision, and then figures out how the case should be entered into the system to achieve this outcome[1]. The users' ability to perform their work well rests on how good they are at building a coherent understanding of their concrete cases. This understanding enables them to make just decisions, which are subsequently implemented through the use of the CSA system. In complex cases the building of this understanding may involve discussion with colleagues or service consultants at the call center, but once the way to handle a case has been settled the users rarely need further guidance to actually go through the screens and input the data. Thus, whereas the actual operation of the CSA system is a minor issue, the user needs a detailed understanding of how CSA cases are modeled in the system to be able to achieve the desired outcome in simple as well as complex cases.

In this regard, we can distinguish three broad sources of complications in using a system to solve complex tasks:

- *Problems handling the complexity of the work domain*. The CSA system can take over most of the calculations and bookkeeping but it cannot do away with the essential difficulties involved in CSA work. Rather, these essential difficulties are what constitute CSA work, and the users are prepared to be spending their time grabbling with them.
- *Problems operating the system*. It is both possible and commendable to choose sensible labels, avoid random inconsistencies in the user interface, and in other ways make the system easy to operate. The operation of a system is, however, only a minor part of using it, and ease-of-operation cannot make up for the more profound complications.
- *Problems mapping between the real world and the system*. When systems get more complex and still closer to the users' work, a consistent and transparent mapping between the real world and the system becomes crucially important. To be usable the CSA system must enable its users to readily predict the real-world outcome of the various system facilities vis-à-vis the users' concrete cases.

[1] Note that this has nothing to do with circumventing the rules; it is entirely about competent use of tools.

The primacy of a consistent and transparent mapping between the real world and the system is well supported in the literature [e.g., 1, 6, 11] and means that the task gap introduced by reusing functionally rich components is a critical issue. Users need an intelligible system image that is consistent with their work tasks but they are, instead, likely to get a system image that is composed of a set of related but not fully integrated component images. This introduces additional dimensions into an activity the users already experience as difficult, and in a certain sense these additional dimensions are foreign to the users' work. Users expect new systems to give them more time for their work or make them more capable of accomplishing it, and they are only prepared to spend a limited amount of their time and attention on making sense of a computer system. Consequently, systems development organizations should be wary not to uncritically adopt techniques that support reuse at the expense of more fragmented system images.

7. Awareness of CBD's effect on usability

In the service consultants' opinion, the designers/developers often display a somewhat shallow and simplistic understanding of what the users need and what cause them trouble. On the one hand, the designers/developers are biased toward the system-centered issues that form the bulk of their work:

> Service consultant: But what you [a designer/ developer] don't know – or do not think about – that's *use*. There is no doubt that you're the one person who knows most about how the existing CSA system is composed.

On the other hand, the service consultants often find it frustratingly difficult to communicate that the user-centered issues are much more about bridging the gap between the real world and the system than about operating the system.

The service consultants find that the way the designers/developers interact with the users is one reason for their somewhat shallow understanding of user issues. Whereas the service consultants are called by users who are in the midst of their work and experience a problem or an exceptional case, the designers/developers interact with users through relatively brief encounters where the users are taken out of their work context and interviewed about a set of issues. These interviews supplement a series of full-day meetings between a selected group of about ten users and a group of CSA engineers, including both designers/developers and the two service consultants. The basic problem with the interviews is that they end up focusing on mainstream cases at the expense of a number of exceptions that should also be covered by the system:

> Service consultant: It is a problem, though, that when you [the designers/developers] go out and talk with them [the users] they primarily think of all the ordinary things.
>
> Designer/developer: Of course they do.
>
> Service consultant: Often, they don't think about the ah-then-there-is-also and yeah-that's-also-possible cases.

When a designer/developer for example asks whether a specific facility would be useful, the users often base their reply on the mainstream cases, which are numerous but relatively easy to handle. Sometimes the designers/developers get positive feedback on facilities that are too simplistic to handle the exceptions, which may be few in numbers but take up a lot of time and resources. In these cases it is up to the service consultants to spot that further analysis is required. These cases also illustrate that the interviews are likely to give the designers/developers another view on what constitute the principal usability concerns in the CSA system than the one the service consultants get from their work in the call center.

Generally, the two service consultants have been more concerned with the consequences of adopting component-based design than have the other CSA engineers. The service consultants spend a great deal of time arguing that the users primarily experience problems with regard to handling the complexity of their work domain and mapping between the real world and the CSA system. The adopted approach to software reuse is based on turning existing systems into functionally rich components and this seems to add new complexities to the mapping between the real world and the CSA system. To the users this is a potentially severe consequence of a type of component-based design that is considered promising by development organizations.

8. Conclusion

Component-based design, a relaxed approach to object-oriented design, is gaining attention as a promising technique for organizing and accomplishing software reuse. From a practical point of view, an important part of this reuse potential lies in the possibility of turning existing applications into reusable components and, thereby, preserving the investment in legacy systems. Component-based design is generally considered a technique of interest to development organizations as a way to save resources, which may then be allocated to other activities, and provide more comprehensive functionality, which would otherwise be too costly to (re)develop. The consequences for users have received comparatively little attention, except for pointing out that component-based design entails increased interface consistency.

Based on fieldwork in a software development company where existing applications are turned into functionally rich components, this study analyzes how the consequences of component-based design reach beyond the development process and well into system use. The studied development project is to completely redevelop a system, which is intended to assist its users in performing a complex work task. In addition, the system is to model this task at a very fine-grained level. It should be noted that this study is based on data from the development process; the studied system is still under construction and thus no data on real use are available. While no strong claims can be made as to the generality of the findings, it is reasonable to assume that they will also be applicable in other settings where complex systems are assembled from functionally rich components.

The usability of a system for a complex work domain is critically dependent on a system image that is intelligible and consistent with the users' work tasks. This study shows that when systems are assembled from functionally rich components the users are, instead, likely to get a system image that is composed of a set of related but not fully integrated component images. The components incorporate assumptions about the users' tasks, and because the components have different origins these assumptions differ across components. This adds to the complexity of using the system because it is left to the users to bridge the gaps between the components. Consequently, the use of components makes it more difficult to work out the real-world meaning and consequences of the various system facilities. In the studied project this potential degradation of system usability was gradually realized and discussed in terms of (1) gaps between the system and the user's task, (2) conceptual mismatches across components, (3) the need to rekey information that had already been entered into another component, (4) scalability problems when components were used for other purposes than those originally envisaged, and (5) the need for increased education and training. It is anticipated that these issues get increasingly severe as systems are developed to handle increasingly fine details of the users' work.

While development organizations may potentially reap a major payoff from component-based design, the users may experience a degradation of system usability. This calls for reflection on how previous experience and work products can most fruitfully be brought to bear on new development projects.

9. Acknowledgements

This work has been supported by a grant from the Danish National Research Foundation. I wish to thank the members of the CSA project group who have put up with my presence in spite of their busy schedule.

10. References

[1] Bannon, L.J. and Bødker, S. "Beyond the interface: Encountering artifacts in use", in Carroll, J.M. (ed.) *Designing Interaction: Psychology at the Human-Computer Interface* (pp. 227-253), Cambridge University Press, Cambridge, 1991.

[2] Biggerstaff, T.J. and Richter, C. "Reusability framework, assessment, and directions", in Biggerstaff, T.J. and Perlis, A.J. (eds.) *Software Reusability: Concepts and Models* (pp. 1-17), ACM Press, New York, 1989.

[3] Brooks, F.P. "'No silver bullet' refired", in Brooks, F.P. *The Mythical Man-Month: Essays on Software Engineering* (pp. 207-226), Addison-Wesley, Reading, MA, 1995.

[4] Curtis, B. "Cognitive issues in reusing software artifacts", in Biggerstaff, T.J. and Perlis, A.J. (eds.) *Software Reusability: Applications and Experience* (pp. 269-288), ACM Press, New York, 1989.

[5] Grinter, R.E. "Recomposition: Putting it all back together again", *Proceedings of the ACM CSCW'98 Conference on Computer Supported Cooperative Work* (pp. 393-402), ACM Press, New York, 1998.

[6] Grudin, J. "The case against user interface consistency", *Communications of the ACM* (vol. 32, no. 10), 1989, pp. 1164-1173.

[7] Hertzum, M. "People as carriers of experience and sources of commitment: Information seeking in a software design project", to appear in *The New Review of Information Behaviour Research: Studies of Information Seeking in Context. Proceedings of ISIC 2000* (Göteborg, Sweden, August 16-18, 2000).

[8] IDC. *Managing component-based development: SELECT software tools*, International Data Corporation, Framingham, MA, 1998. Available at: http://www.selectst.com/downloads/IDC/IDC.asp (consulted March 29, 2000).

[9] Leith, P. "Fundamental errors in legal logic programming", *The Computer Journal* (vol. 29, no. 6), 1986, pp. 545-552.

[10] McClure, C. "Experiences in organizing for software reuse", Extended Intelligence, Chicago, IL, 1995. Available at: http://www.reusability.com/papers3.html (consulted October 10, 1999).

[11] Norman, D.A. "Cognitive engineering", in Norman, D.A. and Draper, S.W. (eds.) *User Centered System Design: New Perspectives on Human-Computer Interaction* (pp. 31-61), Lawrence Erlbaum, Hillsdale, NJ, 1986.

[12] Ockerman, J.J. and Mitchell, C.M. "Case-based design browser to support software reuse: Theoretical structure and empirical evaluation", *International Journal of Human-Computer Studies* (vol. 51), 1999, pp. 865-893.

[13] Parnas, D.L.; Clements, P.C. and Weiss, D.M. "Enhancing reusability with information hiding", in Biggerstaff, T.J. and Perlis, A.J. (eds.) *Software Reusability: Concepts and Models* (pp. 141-157), ACM Press, New York, 1989.

[14] Sprott, D. "Componentizing the enterprise application packages", *Communications of the ACM* (vol. 43, no. 4), 2000, pp. 63-69.