Second International Workshop on Component-Based Software Engineering

Position Paper: David Budgen

Department of Computer Science
Keele University
Staffordshire
ST5 5BG
U.K.

1. Comments on the CBSE Strawman Document

My (brief) comments on this are as follows:

- The idea of exploring the issues via a Strawman document such as this seems a very good one, and I think that producing such a document forms an excellent step in the development of the area.
- I would argue that the document cannot afford to ignore the problems of *creating* such systems and that there is a need for a section that addresses this, even if at the present, we have only limited experiences to offer. I have suggested the outline for a new section below (probably it should become a new section 3, with the present 3 becoming 4). As a part-justification for this, I would observe that this is identified as a significant theme within the PITAC Interim Report (http://www.ccic.gov/ac/interim/), which comments that "Special emphasis should be placed on component based software design and production techniques" in identifying research priorities.
- The other area that has emerged as needing more attention from some work of our own, that has involved making a small investigation into *how* people design solutions from components, is that of component *documentation.* This is an area where there appears to be a potentially key need for standard forms of description to be developed.

2. Suggested structure for a new section

System development for CBSE-based projects
- Life-cycle issues
  - Integration with 'traditional' waterfall activities (req. elicitation; design; testing)
  - Role in Rapid Application Development
- Design Practices
  - Reuse strategies (horizontal/vertical)
  - Changes to practices used in system design
  - Underpinning technology needs /* reason to precede existing section 3 */
  - Component development practices
- Testing and Evaluation
  - Recommended practices
  - Documentation

3. Discussion of design practices for CBSE

At present no widely recognised software design practices incorporate the concept of reusing pre-existing components. The 'historical' approach to software development has (on paper at least) encouraged the adoption of design methods such as SSA/SD, JSD, OOSD etc., which assume:

- A 'clean slate' solution, in which each design solution is custom-built to fit the needs of the particular problem.
- The reuse of design 'process' experience is mainly conveyed through the practices of the design method.

Indeed, some of the current interest in the concept of 'design patterns' [Gamma et al., 1995] may well reflect the difficulty that these existing practices have in incorporating guidance on reuse.

Support for the idea of software component reuse in design needs to be related to the concept of software architecture. In the past, each system has been custom built, and its architectural form has been determined on the basis of such considerations as reuse and locally preferred practices, or as a separate and independent choice. This is also an area where the experience of hardware development has been very positive, in that the adoption of standard architectural forms has formed an important underpinning to component reuse. For software the concept is still in relatively formative stages [Shaw and Garlan, 1996; Monroe et al., 1997], and the ease with which software can be modified also makes it harder to enforce any standards for software architecture.

Historically the main corpus of software design literature has been focused upon the task of designing 'bespoke' solutions, although it can be argued that use of a design method produces an 'architectural style' that increases the potential for reusing elements of designs produced using the same practices. However, th
is has traditionally received little attention and we are not aware of the existence of any widely used systematic practices that are based upon reuse of part-solutions.

Observations of software designers and their practices in a 'bespoke development' context have indicated that such reuse of part-solutions does occur [Adelson and Soloway, 1985; Guindon, 1990], although during the design process this may be more a case of reusing paradigms or idioms from previous experience (termed 'labels for plans'), rather than involving actual system elements.

These notes review the question of how a transition to a component-based paradigm is likely to alter the designer's goals and the overall system expectations. We then examine some of the empirical work on software design activities as a preliminary to considering how design practices will need to change to meet these goals. Finally, we make some brief comments based upon some recent work of our
own.

Changing the goal posts

The distinction between our expectations when producing bespoke software solutions and reusing components should perhaps be similar to our expectations of ‘made to measure’ and mass-produced clothing. It we have a suit ‘made to measure’, then we expect it to be not only a good fit (and many people are physically asymmetrical as regards such issues as shoulder height and arm length), but also to be able to provide for any personal preferences such as secure inner pockets. However, if purchasing ‘off the peg’ clothing then we are prepared to accept some degree of compromise providing that the overall fit and serviceability is acceptable (and of course, that the cost is also significantly less than that of the bespoke product). Should it become necessary to alter the off the peg product to achieve an acceptable fit, the question of cost becomes critical, since this may approach or even exceed the cost of purchasing a bespoke garment.

Within the software development process there are clear analogies to this, and in order to achieve an equivalent degree of compromise when proposing to develop a solution using components, we need to be able to determine both:

- what degree of compromise we can accept
- how we are to measure this

and to agree these ’up front’ before beginning the processes of design and development. Indeed, it may even be that a thorough analysis will reveal that a component-based solution is not necessarily more cost-effective than a bespoke development. (There is an interesting caution to this effect in the conclusions of [Lewis et al., 1992], where they observe that what they term ‘strong reuse encouragement’ may lead to "the subject’s reuse of inappropriate components", resulting in lower productivity!)

Unfortunately, the invisible and multi-faceted nature of software makes it much harder to find criteria for compromise than is the case for clothing! Some possible parameters that a designer might need to consider (for both the overall solution and also in some cases, for the components) include:

eventual system performance
resource use (especially memory)
interface forms
functionality

(Note that these are not necessarily independent.) These factors will also have implications for the ways that components may need to be documented.

Empirical studies of designers

The corpus of material covering empirical studies of software design activities is a fairly small one, generated chiefly by workers with a background in cognitive psychology as well as software. A significant feature of this is the relatively small number of subjects used by most researchers, reflecting the difficulty of gaining access to suitable experienced software designers, and even of suitably inexperienced student subjects, since even inexperienced subjects need knowledge about design.

We review this material here mainly in terms of the extent to which any component-related concepts can be identified. (Since none of it was originally concerned with such systems, we should not regard this as necessarily providing an exhaustive analysis!)

One of the earliest published studies on observations of larger-scale software design activities is that of [Adelson and Soloway, 1985], which examined the strategies used by a small number of experienced designers (three) plus two novice designers, when faced with problems that could be categorised as follows.

- familiar problems in a familiar domain
- unfamiliar problems in a familiar domain
- unfamiliar problems in an unfamiliar domain

Two of the important ideas that they identified in terms of our own focus of interest was that designers frequently adopted an opportunistic strategy (which we can characterise as being problem-driven rather than method-driven) in solving their problems, and the concept of labels for plans. The latter can be considered as a means of identifying where previous experience can be reused, by noting the existence of a previously-used plan for a part-solution, which can be retrieved from memory without the need to work this through in detail while solving the current problem.

Later work by Guindon was based upon a study of three experienced designers, and the work reported in [Guindon, 1990] focused on investigating the means by which the subjects exploited their prior knowledge and experience in solving a problem. In particular, this study identified the use of schemas by designers to aid reuse of experience, where "a schema is a complex rule composed of a pattern which specifies the similarities in requirements between different instances of a class of systems". Such a description of a schema can probably be loosely equated to the concept of 'labels for plans' described by Adelson and Soloway. It was also noted that "designers tend to reuse the same successful solutions over and over in their career".

In terms of a component-based approach, both of these studies do indicate that experience is reused at an abstract level, although not necessarily mapped on to concrete components. However, since components were not included in the studies, this is perhaps not surprising.

A very thorough analysis of both the nature of the design process as applied to software, and also of the methodological issues that arise when conducting any study of how software is designed is provided in [Davies and Castell, 1992] . A similar analysis in [Visser and Hoc, 1990] contains one of the few references to problem of reuse that we could identify, and concludes that:

> "*one may think that most psychological studies paying so little attention to this reuse - contrary to software engineering, which considers it a major problem to be solved - is due to their limited context making reuse difficult to implement*"

Modifying the design process

Having argued that the 'traditional' approach to software development based upon design methods that produced bespoke solutions does not provide adequate support for a component-based development strategy, and identified that the bulk of empirical studies do not address the component concept, one question that arises is that of how such systems might be most effectively developed.

Two identifiable strategies, based on different criteria, are described below. We should note that these probably describe fairly extreme forms, and that any practical development is likely to involve some degree of compromise between the two.

1. Assuming that an architectural style is already established, for whatever reasons, then the components need to be identified on the basis of their ability to conform to the needs of the architecture as well as on their functionality. We can regard this strategy as one of **Framework First**, using the pre-determined framework to help narrow down the search space for components. Certainly such an architecture will itself have some specific characteristics, and is likely to imply some form of open, loosely coupled interaction, including the possibility of 'plug and play' capability to support evolution.

2. A quite different approach is one that begins by identifying a set of suitable components and then to decide upon the architectural form for the solution on the basis of how to get the best from the selected components. Again, such an **Element First** approach is more likely to imply a more closely coupled strategy, and may provide less scope for long-term evolution.

Framework First The adoption of such a strategy is consistent with the needs of larger organisations, concerned with reuse of internally generated components as well as those obtained externally, and with longer-term maintenance needs also in view. As with adoption of design methods, such a strategy does require careful consideration before choosing the framework, since this decision occurs both early in development (or in transition to a component-based philosophy) and may also have significant longer-term effects. (While the Object-Oriented paradigm is probably the predominant one currently in use for new developments, it may not always be the most appropriate framework, especially where real-time needs predominate.)

Implicit in this choice of strategy must be an acceptance that it may be necessary to develop either new components, or modifications to existing ones, in order to achieve a solution for a given problem. This represents a major risk factor, and is also strongly linked to the choice of framework.

Element First For this strategy to work, the designer may need to be able to search a very large solution space in order to find appropriate components [Frakes and Pole, 1994]. Its adoption is therefore much less one of an organisational one, and as we identified above, it is probably a strategy that is only really suited to solving 'one-off' problems, with relatively little need to consider longer-term maintenance needs.

The main problem (or risk factor) is that of making an adequate search for components at the

outset. This requires both a wide search space and also efficient methods of searching this. Given that component classification is likely to be uneven, ensuring that any search is made effectively is likely to present the major risk, since an eventual set of components that have mis-matched interfaces are unlikely to provide an overall solution that is satisfactory in terms of the parameters identified above.

A second issue that arises is the degree of modification that is acceptable [Basili et al., 1994]. Since modification is likely to be expensive; require expert skills; introduce the need for additional version management; and increase testing needs, it is clearly an option that requires care. The idea of the wrapper [Brown and Short, 1997] is probably a more attractive approach.

Observations from some simple empirical studies

We have undertaken some simple studies of design activity, based upon the use of Unix processes as reusable components, with some intermediate findings being described in [Budgen and Pohthong, 1999]. Since the choice of shell programs effectively constitutes an element of 'Framework First', we have been mainly concerned to investigate the search and selection strategies used by our subjects in a series of studies. (We might also observe that the whole philosophy of Unix also means that Unix utilities are effectively designed for the purpose of reuse.)

One significant issue that has emerged is the designer's need for*information* about a candidate component, regardless of the strategy used to find it. On some occasions, this has also included executing the component separately, in order to investigate its behaviour when used in a particular mode. While these observations do need to be further investigated in a wider context, they have significant implications for both the design of tools intended to support component-based development and also for any documentation standards that might emerge.

References

[Adelson and Soloway, 1985] Beth Adelson and Elliott Soloway, The role of domain experience in software design, IEEE Transactions on Software Engineering, 11(11), November 1985, 1351-1360.

[Basili et al., 1994] V R Basili, L C Briand and W I Melo, How Reuse influences productivity in Object-Oriented Systems, Communications of the ACM, 37(5), 1994, 104-115.

[Brown and Short, 1997] Alan W Brown and Keith Short, On Components and Objects: The Foundations of Component-Based Development, Proceedings of the 5th Int. Symposium on Assessment of Software Tools and Technologies, 1997, 112-121.

[Budgen and Pohthong, 1999] David Budgen and Amnart Pohthong, Component Reuse in Software Design: An Observational Study, submitted for publication.

[[Davies and Castell, 1992] S P Davies and A M Castell, Contextualizing design: narratives and

rationalization in empirical studies of software design, Design Studies, 13(4), 1992, 379-392.

[Frakes and Pole, 1994] W B Frakes and T P Pole, An Empirical Study of Representation Methods for Reusable Software Components, IEEE Transactions on Software Engineering, 20(8), 1994, 617-630.

[[Gamma et al., 1995] E Gamma, R Helm, R Johnson and J Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley 1995.

[Guindon, 1990] Raymonde Guindon, Knowledge exploited by experts during software system design, Int. Journal of Man-Machine Studies, 33, 1990, 279-304.

[Lewis et al., 1992] J A Lewis, S M Henry, D G Kafura and R S Schulman, On the relationship between the object-oriented paradigm and software reuse: an empirical investigation, Journal of Object-Oriented Programming, 5(4), 1992, 35-41.

[Monroe et al., 1997] R T Monroe, A Kompanek, R Melton and D Garlan, Architectural Styles, Design Patterns, and Objects, IEEE Software, January 1997, 43-52.

[Shaw and Garlan, 1996], Mary Shaw and David Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice/Hall, 1996.

[Visser, 1987] Willemien Visser, Strategies in Programming Programmable Controllers: A Field Study on a Professional Programmer, in Empirical Studies of Programmers: Second Workshop, Editors G Olson, S Sheppard & E Soloway, Ablex Publishing, 1987, 217-230.

[Visser and Hoc, 1990] W Visser and H-M Hoc, Expert Software Design Strategies, in Psychology of Programming, Editors J-M Hoc, T Green, R Samurçay and D Gilmore, Academic Press, 1990, 235-249.