

**BRIDGING INDUSTRIAL DESIGN AND SOFTWARE ENGINEERING
THROUGH CONSTRAINTS IDENTIFICATION, SOLUTION SPACE
OPTIMISATION AND REUSE.**

BESNARD, Denis

Denis.Besnard@ncl.ac.uk
tel. 00 +44 (0) 191 222 8058
fax. 00 +44 (0) 191 222 8788

&

LAWRIE, Anthony, T.

A.T.Lawrie@ncl.ac.uk
tel. 00 +44 (0) 191 222 6858
fax. 00 +44 (0) 191 222 8788

Department of Computing Science, Centre for Software Reliability
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
UNITED KINGDOM

Abstract: Design is a complex activity that can be analysed from a wide variety of perspectives. This paper attempts to get into the details of this particular problem solving process, taking into account psychological arguments. We characterise some of the phases involved in the design process, namely the constraints identification, the optimisation of solution space and the reuse process. We highlight a three-dimensional framework of how the constraints identification impacts on the solution space which, in turn, determines the range of the components that will be eligible for reuse. We discuss this argument through examples from both inside and outside the software engineering field. We then consider how our conception can describe successful designs and conclude by briefly looking at reuse as a general design policy.

1 Introduction

Although many useful analysis and design techniques have been pioneered in software engineering's relatively short history (see Bell, 2000; Pressman 1992; Sommerville, 1992), its inherently conceptual nature (Brooks, 1995) ensures that software development continues to retain a significant 'craft-based' element for individual creativity and skill (Tekinerdogan, 2000). These intangible attributes set it apart from most other engineering domains where eventual software failures often result from discrete human design errors made early in the development life-cycle (Health & Safety Committee, 1998). Nevertheless, it has still been argued that important lessons can be learned for software engineering from more established and mature classical engineering disciplines (Leveson, 1994; Holloway, 1999). This is the philosophy of our paper, also.

In respect to these individualistic and highly conceptual elements of software engineering, this paper focuses attention upon individual cognition and the potential for human cognition in optimising the design of software in computer-based systems. This involves drawing upon well established psychological research to emphasize the importance of constraints identification and its resulting influence upon solution finding and knowledge reuse during the design phase. To exemplify our conceptions, we draw upon both successes and failures in traditional and software engineering domains.

2 Design: a cognitive activity approach

Designing can be approached as a problem solving activity (see Sommerville, 1992; Tekinerdogan, 2000), and analysed insightfully from a cognitive perspective (see Brooks, 1999; Von Maryhauser & Vans, 1995; Westerman *et al.*, 1997) Here, it mainly consists in discovering a solution that addresses a design objective, namely, by:

- a) identifying the constraints imposed by both explicit requirements and knowledge assumptions;
- b) seeking options in a solution space;
- c) investigating the possibilities for reuse of previous solutions.

From a cognitive standpoint, design implies achieving a potentially fuzzy goal (Whitefield, 1990) that admits a variety of solutions (Burkhardt & D tienne, 1994). Under this angle, designing implies finding paths in a solution space¹, the latter being defined as the total number of possible intermediate steps that exist between the statement of the problem and its solution (Cordier *et al.*, 1990; Newell & Simon, 1972). Solving a design problem is particular in the sense that the solution space can be enormous. However, some solutions to the current problem may already exist. So there is justifiably great interest in attempting solution reuse, for cost or time reasons (McDermid, 1992).

In this paper, we adopt a human-centred view about design. Cognitive psychology provides a rewarding theoretical framework to explore problem solving where we revisit the activity of design and the three dimensional framework mentioned above. Constraints identification bound the solution space, which, in turn, dictates what components should be selected and reused. Obviously, because of our individualistic psychological view of design, we do not consider other important influences (e.g. process feedback, organisational; see respectively Gilb, 2000; Reason, 1995). We will nonetheless show that an erroneous identification of the constraints strongly influences the failure or success of the design process. Eventually, it can cause the solution space to be narrowed down or widened, introducing unviable design options to be considered or viable options to be disregarded.

¹ Psychologists usually prefer the term *search space*.

3 A 3-dimensional approach of design

We have identified three phases in design where some discussion can improve understanding. These three topics will be investigated from an individual perspective. The next three subsections will address constraints identification, optimisation of the solution space, and solution reuse. In doing so we will attempt to show how the possible linearity involved defines the solution space and how the latter ultimately influences the reuse process.

3.1 The identification of the explicit constraints imposed by requirements

In our view, a set of constraints is the outcome of a designer's interpretation of the requirements, in terms of design implications at the technical level. Accurately identifying the constraints at early stages of the design process is of major importance as it impacts on the solution space (see section 3.2). If one introduces invalid constraints for the realisation of an artefact, the solution space will be narrowed down exaggeratedly, leading to a disregard for viable options. Conversely, if all the valid constraints are not identified, then the solution space erroneously widens, introducing unviable options. In order to give a concrete view of these design conceptions, we will now consider an engineering example: the Tacoma Narrows bridge, in the USA.

In 1940, this bridge had to establish a 2800-foot road link above Puget sound. Due to the strong winds present in the Narrows, the Washington Department of Highways had proposed a suspension bridge with a 25-foot-deep truss along the roadway, for a construction cost of \$11 million. However, two engineers, Leon Moisseiff and Fred Lienhard, had put into practice a new mathematical theory (the deflection theory) for calculating loads and wind forces for suspension bridges. This new theory allowed them to reduce the amount of stiffening material from 25-foot trusses to 8-foot girders. The construction costs dropped to \$6.4 million and this design solution was adopted. However, the novel design caused the bridge to be excessively flexible and despite the checking cables that were added to it after its opening, it collapsed some 5 months later. The investigation of the cause of the failure concluded that ignorance of the actual dynamic effects of wind loads was a significant factor in the accident. Even if some bridges whose design relied on this theory were still standing, it was unsuitable for bridge building in the context of the Tacoma Narrows.

The constraint that was not accurately identified was the degree of influence of wind loads on the bridge structure. It caused the designer to consider the deflection theory as a viable design option. Crossing boundaries is an important part of the design process (Marshall, 2000). However, this crossing must be performed with full awareness. If not, then envisaging a solution which is beyond the boundaries of the underlying theory becomes possible (Holloway, 1999).

On the human side, the best-known cause of human cognition failure is the complexity of a problem (Amalberti, 1996). However, human error can also occur because some important data has been disregarded in solving the problem. The latter is well-known in diagnosis and troubleshooting by doctors (Schanteau, 1992) or electronics and mechanics operators (Besnard, 1999a, 1999b, 2000). Disregarding data is relevant to the design phase also. Bonnardel and Summer (1996) asserted that experienced designers may forget to consider certain criteria for assessing features from a different perspective. Psychologically, this error is underpinned by the designer activating an experience-driven knowledge base (a *schema*; see Reason, 1990) that does not accurately reflect the actual problem (Reason, 1987; Schanteau, 1992). When this occurs, the solution space may be wider or narrower than what is technically optimal, leading the designers to search for solutions in a set that comprises either unviable options or excluding the discovery of viable alternatives. It is therefore important to emphasize the critical role of the identification of the constraints from the initial set of explicit requirements.

In this subsection, we have exposed our ideas about constraints and how errors in their identification could be accounted for by a simple cognitive framework. We are now going to

show how constraints are possibly linearly linked to the solution space. Our aim will now be to demonstrate that dropping or adding constraints in the design setting (because of an erroneous constraints identification process) can impact on the optimisation of that solution space. This position will be framed using the graphical version of the linear programming technique. It will be used to highlight the linear function linking design constraints and the solution space together.

3.2 Optimising the solution space.

Linear programming (LP) is a quantitative problem solving technique that can be found in many mathematical texts (see Lucey, 1992) and often included also in management decision-making literature (see Cooke & Slacke, 1991). The technique is concerned with the quantitative optimisation of an objective when the decision variables are subject to some explicit and quantifiable constraints. The purpose of the following LP scenario is not to advocate its use in software design decision-making, nor do we wish to imply that software design decision-making is strictly linear, since the highly conceptual nature of software development has long been considered as containing non-linear characteristics (Brooks, 1995). Instead we use the graphical version of the LP technique to clarify, both visually and numerically, the role of constraints identification in optimising the solution space and guiding subsequent design reuse. Only the pertinent details are included in the body of the paper. A full mathematical breakdown of the scenario is provided in the Appendix. Interested readers should consult Lucey (1992, chapters 17-20) for a more complete coverage of using the LP technique.

Consider the following scenario: Imagine a software company that manages its software development operations along product lines and product families to maximize software component and source code reuse. A software engineer, as design authority, is assigned to lead the technical development of the new software product. This new product requires both novel and replicated functionality features. The explicit requirements govern the designer's initial investigation and analysis of existing family related products to identify related features that will provide a good design basis. After this initial analysis, the designer identifies three related product components as reuse candidates to begin with.

Because this scenario involves the optimisation of only two decision variables (i.e. design constraints and potential reuse solutions) the graphical method of linear programming can be legitimately used. The LP graph below (Figure 1) maps the linear and fixed limits to illustrate the feasible region of the solution space for the LP scenario just described. The fixed straight lines (D, E & F) illustrate the explicit design requirements that were originally prioritised for the new software product. The diagonal lines (A, B & C) represent the linear relationships that exist between the design constraints and design solutions for the three reuse candidates. Collectively, they define the feasible solution region for the design problem set.

The LP graph takes into account the optimisation of the feasible solution region by identifying the maximal number of design constraints that still permit the maximum number of viable solutions to be considered. How to identify this optimal solution point is clearly represented. It is achieved by intersecting the rightmost diagonal line at the mid-point of the feasible region from both horizontal and vertical axes at 90°. From the graph, this is when 25 design constraints are identified, giving 37 viable reuse solutions².

² Please note that this technique is simplified here because none of the decision variables involved contain coefficient factors.

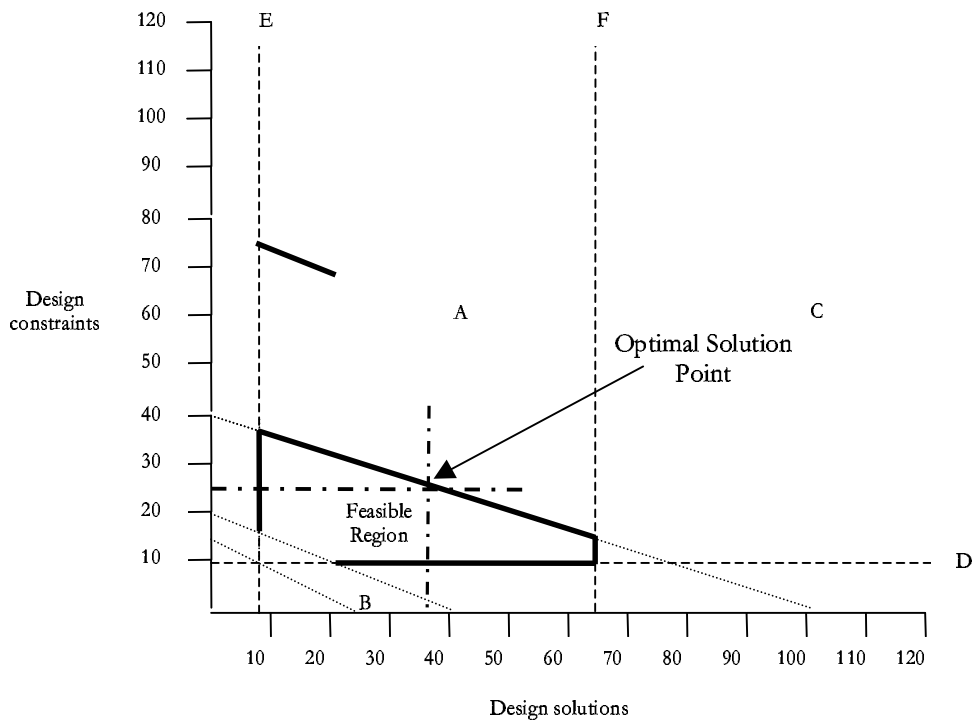


Figure 1: Determination of the optimal solution point with the graphical linear programming method.

The graph in Figure 1 has some relevance with regards to the transition from the identification of constraints to the choice for solution reuse. Let us assume that the diagram reflects the optimal solution space region for the design scenario set (i.e., in our case, 25 constraints identified and 37 possible design reuse solutions). Then if 5 valid design constraints are overlooked, this will increase the number of potential solutions from 37 to approximately 50, admitting 13 unviable reuse solutions. This sub-optimisation of the solution space could result in the adoption of unsuitable design reuse solutions that later fail during operation when the effects of critical constraint omissions are experienced. Equally, admitting redundant constraints can also have a negative design effect. For example, if 5 invalid design constraints are erroneously included, the subsequent solution search will preclude 10 viable potential solutions for reuse or adaptation, reducing the number of potential solutions from 37 to 27. This sub-optimisation of the solution space could result in unnecessary reinvention (and its consequent increased costs). It can even become a barrier to innovation through the designer(s) believing that the design problem set is unsolvable³.

The linear programming approach allowed us to demonstrate how disregarding or adding constraints affects the solution space. This visual example helps the conceptualisation of the intuitive notion of the optimal point between design constraints and solution options. It allows the designer to be aware of the important trade-off and compare several components against each other during the process of solution reuse. As we have stated earlier, in section 3.1, the

³ Although we adopt an idealised view about solution optimisation, we are aware that constraints are usually open to negotiation. Relaxing some of them in order to voluntarily widen the solution space, in practice, is always a distinct possibility. In this respect, we only take a snapshot and preclude temporal change considerations.

constraints impact on the optimisation process of the solution space. We have now exposed how the aim of optimising the solution space could be established as a meta-design process objective in its own right. The next section will document and discuss the catastrophic effects of deriving sub-optimal solution regions and the selection of subsequent design reuse in software engineering. We will also briefly consider the fundamental psychological concepts involved.

3.3 Reuse

In software engineering, reuse is common practice⁴. Among other things, it is aimed at reducing costs and development time (Richards, 2000). McClure (1992) asserts that most of the code developed for an application is reusable in other applications, because only small proportions are program-specific. Nevertheless, software reuse must not be conceived as an immediate cut-and-paste from one program into another, as it is even unlikely that any component can be completely reused so readily (Sommerville, 1992). Rather, the decision for reuse is supported by a compromise between the adaptability of the desired artefact, the cost of restructuring it (Buratto & Chabaud, 1994) and its context of reuse. Consequently, poor reuse selection can frequently result in the cost of adapting an existing piece of software being greater than the cost involved in developing it from scratch (Pressman, 1992). These drawbacks of software adaptability are well known to software programmers and designers alike. They have become well established for a long time in the literature (see: Weinberg, 1971). However, the cost and time-saving benefits of reusing software can prove so beneficial in large software engineering projects that the actual software design process may become overly reuse-driven. In some cases, such over reuse of software can result in catastrophic operational failure. The case of Ariane 5 explosion will now be discussed from this perspective.

On the 4th of June 1996, 42 seconds after take-off, Ariane 5 veered off its trajectory to such an extent that the on-board self-abort system triggered, causing the complete loss of the launcher. A piece of software (the inertial reference system), dedicated to keeping the launcher on trajectory during the early phase of the flight, was reused from the previous launcher Ariane 4. Due to Ariane 5 having a different behaviour on the first seconds of the flight, unusual horizontal velocity values were generated. One of the errors identified resulted from some a) modules of the previous program not being tested for exceptions when reused and b) in the faulty belief that the safety margin was large enough so that the software could handle the new values (see Lions, 1996 for full details).

Cognitive psychology analyses reuse from the point of view of reasoning by analogy (Burkhardt & Détienne, 1994). In this view, solving a problem is achieved by identifying some similarities between the current problem (the target problem) and one that has been solved in the past (the source problem) (Catrambone & Holyoack, 1989; Novick & Holyoack, 1991; Gick & McGarry, 1992). One of the causes of errors lies in the possibility of not identifying important data in the source problem (Novick, 1988) or neglecting some discrepancies between the two problems' structures. Again, the quality of the schema used by the designer is of relevance here. It may trigger itself in sub-optimal conditions, causing the analogical reasoning to become flawed.

On the other hand, reuse is usually performed with testing and validation procedures, reducing the possibilities of such errors. So, beyond a mere individual explanation of reuse errors, the Ariane 5 accident can be seen as an example of a latent⁵ reuse error. Latent errors have the particularity to remain in a system without any symptom until the conditions needed for them to trigger are present (see Reason, 1995; Randell, 2000). In the case of Ariane 4, the horizontal velocity values could virtually not go beyond the limits present in the software. So, during the

⁴ Although we are aware of *ad hoc* forms as well as systematised forms of reuse, we wish to keep a generalised view on it. In our conception, the cognitive factors involved are generic, transcending the reuse mode itself.

⁵ Computing scientists will prefer the term *dormant* fault.

development of the inertial reference system it was decided that protecting it from being made inoperative by excessive horizontal velocity values was not necessary⁶. It only became a problem when the inertial reference system software was reused.

4 Describing successful designs

So far we have presented our three dimensional view of design in a negative light: the potential for both design errors and barriers to innovation through sub-optimising the solution space. However, this need not be so. In the two cases that follow, we emphasise how our conception of design can explain innovative solution finding also. To maintain a symmetrical balance of cases used we will draw again on bridge building and computer orientated design problems. These examples are the Millennium bridge in Newcastle upon Tyne (UK) and the 32-bit memory architecture of the Eagle computer. In doing so, we will then try to convince the reader that both "classical" industrial design and software engineering design share a similarity that grounds the overarching rationale in this paper.

The Millennium bridge, Newcastle upon Tyne, UK⁷

As is often the case in design, the challenge was finding a novel solution that would compromise optimally all the existing requirements. Tekinerdogan (2000) recognises this as 'innovative design'. The Millennium bridge would explicitly have to:

1. provide cross-river access to pedestrians and cyclists;
2. link the quaysides at only 4-5m above the Tyne level;
3. allow 25m of headroom over a 30m-wide navigational channel;
4. preclude any construction on the quays themselves;
5. show some novelty.

In terms of technical constraints, the access to the bridge by pedestrians and cyclists implied that the bridge had to be built at the road level. But the needed 25m headroom for allowing ships to pass eliminated the possibility of a fixed flat pathway. Moreover, being barred from building anything on the quays themselves and having to provide a novel design surely precluded numerous reuse solutions such as swinging or opening bridges. The solution that the designers found was a curved pathway that would be suspended by another curve. Then tilting them altogether like the visor of a helmet would create enough head room for bigger boats to pass (Figure 2).

Although it is a suitable solution for most bridges, Clarke and Eyre (2000, p. 31) explain that it was impossible to build a straight bridge that would satisfy all the requirements. But if you erroneously continue with such an idea, you will produce a totally different design. You will eventually be forced to drop out some of the initial requirements because of the difficulty to reconcile i.e. a low access and 25m head room.

⁶ Although the consequences of this decision were not fully understood.

⁷ Unless otherwise noted, the source for the material in this section is in Clarke and Eyre (2000).

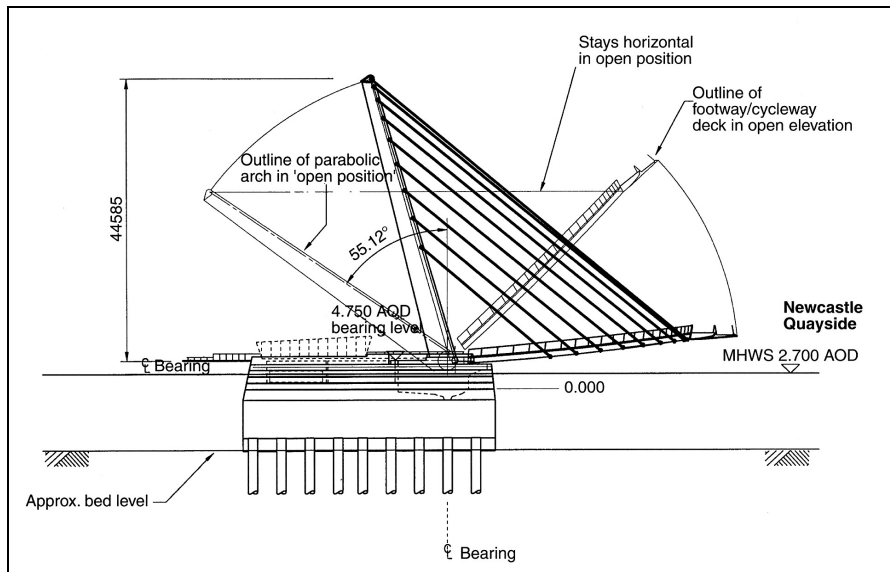


Figure 2: The Millennium bridge. Technical diagram (Clark & Eyre, 2001).

The 32-bit memory architecture of the Eagle computer⁸

The design of the 32-bit Eagle architecture (in the early 1980s) is reported here as a factual case where design success is achieved by trading-off between constraints and reusable solutions. In this case, a single designer had to design one of the maiden 32-bit memory architectures and provide time-sharing security protection. Thus, the design had to integrate these two main constraints, narrowing the solution space dramatically. One possible reuse solution would have been to provide both memory location and security protection using separate security rings and memory addresses. Through critically exploring the various possibilities within the feasible solution space the designer innovatively perceived that the first 3-bits of the memory address could be used to represent both segment and ring number. With such a solution, the memory segment addresses, themselves, would indicate which areas of memory were to be restricted. Moreover, the implementation of this solution would provide 8 levels of security and intrinsically protect memory allocation. The innovative adaptation of established memory protection architecture proved to be a simpler, cheaper, more efficient and more reliable system than any design previously. Again, in Tekinerdogan's (2000) view, this is "innovative" design.

In both cases the task of the designer is to find a way to focus and merge explicit constraints in such a way as to derive an optimal solution. In this respect, two different designers designing two different types of artefacts faced the same inevitable design challenge. In each case they needed to critically explore the feasibility within the explicit constraints set by the requirements. However, more importantly, they were able to remove implicit constraints through questioning the possible validity or adaptability of existing design knowledge. In the case of the Millennium Bridge this was going beyond the design assumption that all bridge designs must be straight. In the case of the 32-bit Eagle architecture it was disregarding the design assumption that memory segments and security rings must be separately represented in the architecture. Once the designers began to question these broad categories of established knowledge, they removed implicit paradigm-fixing constraints and optimised the solution space by widening it to include other innovative and viable solution options. This stands in stark contrast to widening the solution space erroneously through omissions, ignorance, or oversight, as was the case with the Tacoma bridge and Ariane-5. Therefore, having considered comparable design success and

⁸ Unless otherwise noted, the source for the material exposed in this section is in Kidder (1981).

failure cases using our 3-dimensional view, we can now define what we mean by an optimal solution space. This is when there exists no omission and redundancy in the viable constraints set identified. This includes both explicit constraints (set by the customer) and implicit constraints (imposed by flawed human translations of the feasible solution regions). As shown in section 5 below, this provides the maximal degrees of freedom and minimal degrees of restriction in finding an innovative design solution.

5 General discussion

It is believed that design is progressively redirected and refined by integrating constraints imposed by a design option. As Figure 3 illustrates, we think that constraints are spread over a continuum.

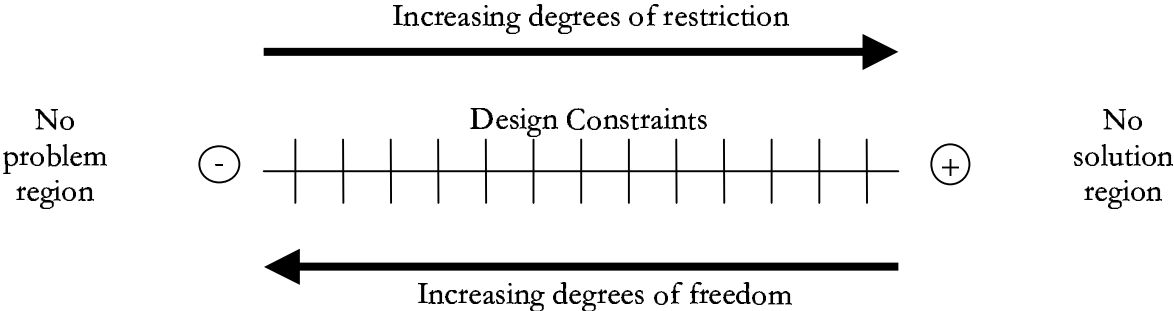


Figure 3: Constraints continuum

In design, as illustrated above, the more constraints there are, the less possible solutions there are, meaning the more you have to trade-off for an exiting solution. The other end of the spectrum is a design region where there is no constraint at all. In such conditions, there is no problem either since the problem is defined by the constraints themselves. Thus, the complexity of a design problem can be thought of in terms of degrees of restriction and freedom allowed by the set of constraints imposed. As the Tacoma Narrows bridge example showed, the errors can be due to allowing too much freedom to your design by neglecting valid constraints. As far as the optimisation of the solution space is concerned, we have exposed our position relying on a linear programming representation. This conception graphically represents the area where possible solutions exist (i.e. feasible region) and what those characteristics (in terms of constraints and solutions) of the optimal solutions are. We think that it is a straightforward and clear way to represent what could remain at a fuzzy and non-verbalisable state in the designer's mind: the concept of the optimal solution space.

Although we have highlighted its potential negative effects, reusing previous solutions is vital in problem solving. Cognitive psychology even considers it as a natural tendency (Roediger, 1980). Reusing components is also the trend in software engineering (See Sommerville, 1992; Pressman, 1992; Bell, 2000). Examples include the building of systems from COTS components, Object-Oriented foundation classes and program language libraries (i.e. C & C++). This reuse trend holds for bridge building too, adding one more similarity between software engineering and "classical" engineering areas. The laws of physics that have been drawn upon the secular activity of bridge building are constantly reused. Moreover, the types of bridges themselves have been categorised, orienting the designer to reuse a building framework, formulas and previous designs when searching for a solution.

Classically, reuse is said to improve control and provide great time and cost savings. As such, it has an economically attractive approach in all engineering disciplines. In commercial development contexts, the design process itself is reuse-driven. However, this policy must not preclude the careful identification of constraints. It is only when the latter have been identified that reuse can have its most powerful and profitable impact on the design process. Considering a component for reuse without having carefully identified the constraints inherent to the concerned design is a mistake for it scales down comprehension of a problem to what the component to reuse can offer. As already shown in the cases provided, a blind reuse-driven design approach can provoke long term losses to be hidden by seeking short-term benefits.

6 Limits

Whilst individual cognition plays a critical role in many engineering disciplines (particularly software engineering), the framework precludes consideration of the many group (Carroll, 1997), political, economic, management and cultural influences (Hollan *et al.*, 2000) that are often identified as significant negative and positive influences of unsuccessful and successful design undertakings. Furthermore, our 3-dimensional framework assumes that explicit requirement constraints can be readily and clearly solicited and understood to begin with. Yet, because of the highly complex and intangible nature of software, the solicitation and communication of requirements has proved to be an uncertain and error prone task in software engineering. In relation to the many design and specification changes that this can cause, our conceptual framework is largely static and does not reflect the many requirements changes that may take place throughout the design phase. Both of these real-world factors would have an uncertain effect upon our conceptual framework, as described in this paper.

Nevertheless, we believe that what we have provided is a kind of meta-design objective that is sufficiently generic to provide the individual designer with some conceptual reference point of the salient factors to be considered and aimed for during the design stage.

7 Further research

Two orthogonal design strategies can be roughly identified. One is constraint-driven. The other is reuse driven. It would be interesting to know if such factors as visibility of constraints or complexity of the problem can influence the balance between these two strategies.

Another potential direction for research is discovering the factors leading a designer to disregard some constraints in the design process. A solution to this human error could then be extremely valuable in avoiding computer-related failures and thereby further improve the dependability of computer-based systems through increased error avoidance (see: Neumann, 1995; Randall, 2000).

8 Conclusion

In concluding this paper it is appropriate to document what value we believe our 3-dimensional design framework adds to the role of design. In answering this we can argue that our paper first seeks to raise awareness of the importance of individual cognition in influencing the success or failure of software design in computer based systems. Secondly, within this scope, we have provided the beginnings of a simple, yet useful, conceptual framework that considers the important and inter-related factors of: constraints identification; solution space optimisation; and design reuse. It is believed also, that this framework is sufficiently generic to act as a meta-conception of what the designer should aim for during the design of both physical and conceptual artefacts. Thirdly, as the design of software systems becomes evermore focused upon processes that build software systems from code-reuse and commercial off-the-shelf components (COTS), the nature of design, itself, will become less create-orientated and even more reuse and selection-orientated. Here, we would argue that we have demonstrated the importance of both explicit and implicit constraint identification in providing a set of conceptual criteria for guiding

this selection and reuse approach. Lastly, we hope by using cross engineering examples and drawing upon multiple disciplines of engineering, psychology, and mathematics for our arguments we have also helped demonstrate the importance and usefulness of an interdisciplinary approach in explaining and exploring individual cognition and its role in classical and software engineering design.

9 Acknowledgements

This paper was written at the University of Newcastle upon Tyne within the DIRC project (<http://www.dirc.org.uk>) on dependability of computer-based systems. The authors wish to thank the following: Prof. Cliff Jones for his influencing initial design comments that helped motivate the writing of this paper: Budi Arief, Jim Armstrong, Cristina Gacek and anonymous reviewers for useful comments; and the sponsors EPSRC for funding this research.

10 References

- Amalberti, R. (1996). *La conduite de systèmes à risques*. Presses Universitaires de France, Paris.
- Bell, D. (2000). *Software engineering: A programming approach*. 3rd edition. Addison-Wesley, U.K.
- Besnard, D. (1999a). *Erreur humaine en diagnostic*. Doctoral dissertation. Université de Provence, Aix en Provence, France.
- Besnard, D. (1999b). Expert error in troubleshooting. An exploratory study in electronics. *International Journal of Human-Computer Studies*, 50, 391-405.
- Besnard, D. (2000). Expert error. The case of troubleshooting in electronics. In F. Kornneef & M. Van der Meulen (Eds). *SAFECOMP 2000*, Springer-Verlag, Heidelberg (pp. 74-85).
- Bonnardel, N. & Summer, T. (1996). Supporting evaluation in design. *Acta Psychologica*, 91, 221-244.
- Brooks, F. P. (1995). *The mythical man month: Essays on software engineering*. Anniversary Edition. Addison-Wesley, New York, NY.
- Brooks, R (1999). Towards a theory of the cognitive processes in computer programming. *International Journal in Human-Computer Studies*, 51, 197-211.
- Buratto, F. & Chabaud, C. (1994). Etude exploratoire du processus de réutilisation de données chez un concepteur d'architecture informatique débutant. In proceedings of *ERGO IA 94*, Biarritz, France (pp. 69-82).
- Burkhardt, J. M. & Détienne, F. (1994). La réutilisation en génie logiciel: une définition d'un cadre de recherche en ergonomie cognitive. In proceedings of *ERGO IA 94*, Biarritz, France (pp. 83-95).
- Carroll, J. (1997). Human computer interaction: psychology as a science of design. *International Journal of Human-Computer Studies*. 46, 501-522.
- Catrambone, R. & Holyoak, K. J. (1989). Overcoming contextual limitations on problem-solving transfer. *Journal of Experimental Psychology: Learning, memory and Cognition*, 15, 1147-1156.
- Clark, G. M. & Eyre, J. (2001). The Gateshead Millennium bridge. *The Structural Engineer*, 79, 30-35.
- Cooke, S. & Slack, N. (1991). *Making management decisions*. Prentice-Hall, UK.
- Cordier, F., Denhière, G., George, C., Crépault, J., Hoc, J.-M., Richard, J.-F. (1990). Connaissances et représentations. in J.-F. Richard, C. Bonnet & R. Ghiglione: *Traité de psychologie cognitive 2*. Bordas, Paris (pp. 35-102).
- Gick, M. L. & McGarry, J. (1992). Learning from mistakes: inducing analogous solution failures to a source problem produces later successes in analogical transfer. *Journal of Experimental Psychology*, 18, 623-639.
- Gilb, T. (2000). The ten most powerful principles for quality in (software and) software organisations for dependable systems. In F. Kornneef & M. Van der Meulen (Eds). *SAFECOMP 2000*, Springer-Verlag, Heidelberg (pp. 1-13).
- Health & Safety Committee (1998). *The use of computers in safety-critical applications*. HMSO, UK.

- Hollan, J., Hutchins, E. & Kirsh, D. (2000). Distributed cognition: toward a new foundation for human-computer interaction research. *ACM Transactions on Computer-Human Interaction*, 7, 174-196.
- Holloway, C. M. (1999). From bridges and rockets. Lessons for software systems. *Proceedings of the 17th International System Safety Conference*, August 1999 (pp. 598-608).
- Kidder, T. (1981) *The Soul of a New Machine*. Back Bay Books, USA.
- Leveson, N. & Turner, C. S. (1993). An investigation of the Therac-25 accidents. *Computer*, 26, 18-41.
- Leveson, N. (1994): "High Pressure Steam Engines and Computer Software." *IEEE Computer* 27, no. 10, 65-73.
- Lions, J. L. (1996). Ariane 5 Flight 501 failure. Report by the enquiry board.
<http://www.cs.berkeley.edu/~demmel/ma221/ariane5rep.html>
- Lucey, T. (1992). *Quantitative techniques*. DP Publications, UK.
- Marshall, L. (2000). Gotos considered taboos and other programmer's taboos. in A. F. Blackwell & E. Bilotta (Eds). *Proceedings of the 12th workshop on the Psychology of Programming Interest Group*, Corenza, Italy. (pp. 171-180).
- McClure, C. (1992). *The three Rs of software automation. Re-engineering, repository, reusability*. Prentice Hall, Englewoods Cliffs, NJ.
- McDermid, J. (1991). *Software engineer's reference book*. Butterworth-Heinemann, Oxford.
- Neumann, P. G. (1995). *Computer-related risks*. Addison-Wesley, New York, NY.
- Newell, A. & Simon, H., A. (1972). *Human problem solving*. Englewood Cliffs, N.J., Prentice Hall.
- Novick, L. R. & Holyoak, K. J. (1991). Mathematical problem solving by analogy. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 17, 338-415.
- Novick, L. R. (1988). Analogical transfer, problem similarity and expertise. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 14, 510-520.
- Pressman, R. S. (1992). *Software engineering. A practitioner's approach*. McGraw-Hill, London.
- Randell, B. (2000). Facing up to faults. *The Computer Journal*, 43, 95-106.
- Reason, J. (1987). A preliminary classification of mistakes. in J. Rasmussen, K. Duncan & J. Leplat. (eds). *New technology and human error*. John Wiley & Sons Ltd, Chichester.
- Reason, J. (1990). *Human error*. Cambridge University Press, Cambridge.
- Reason, J. (1995). *Managing the risks of organisational accidents*. Aldershot, Ashgate.
- Richards, D. (2000). The reuse of knowledge: a user-centred approach. *International Journal of Human-Computer Studies*, 52, 553-579.
- Roediger, H. L. (1980). Memory metaphors in cognitive psychology. *Memory and Cognition*, 8, 231-246.
- Schanteau, J. (1992). How much information does an expert use? Is it relevant? *Acta Psychologica*, 51, 75-86.
- Sommerville, I. (1992). *Software engineering*. Addison-Wesley, Wokingham, UK.
- Tekinerdogan, B. (2000). *Synthesis-Based Software Architecture Design*, PhD thesis, Dept. of Computer Science, University of Twente, The Netherlands.
- Von Maryhauser, A. & Vans, A., M. (1995). Program comprehension during software maintenance and evolution. *Computer*, 28, 44-55.
- Weinberg, G. M. (1971). *The psychology of computer programming*. Van Nostrand Reinhold, London.
- Westerman, S. J., Shryane, N. M., Crawshaw, C. M. & Hockey, G. R. J. (1997). Engineering cognitive diversity. in F. REDMILL & T. ANDERSON (Eds). *Safer Systems*. Proceedings of the 5th Safety-critical Systems Symposium, Brighton, UK (pp. 111-120).
- Whitefield, A. (1990). Human-computer interaction models and their roles in the design of interactive systems. in P. Falzon (Ed). *Cognitive ergonomics: Understanding, learning and designing human-computer interaction*. Academic Press, London (pp. 7-25).

11 Appendix

Mathematical breakdown of linear programming scenario (see section 3.2)

The role of design constraints and solution space optimization is modeled as follows.

Overall design objective= Optimise the solution space through constraints identification

Design objective function $f(c, s)$ = Decision variables of: Design constraints (c)
Design solutions (s)

The Table 1 below shows the designer's initial selection of candidate product family components that provide an initial basis for the prioritized constraints and solutions required for the new product. These are expressed as overriding explicit design requirements of the new product (see limitations D, E & F later marked *).

Table 1: Initial component reuse selection

Component	(A)	(B)	(C)
Constraints (c)	6	8	5
Solutions (s)	3	4	2
Derivative constraints & solutions	$\geq 120 (c)$	$\geq 100 (s)$	$\leq 200 (c)$

The table expresses that component A provides a good basis for satisfying 6 of the prioritized constraints and implementing 3 required solutions with the new product (i.e. **eg?** these maybe quality and performance factors such as portability throughout various other product lines and derived families etc). The bottom row illustrates that the component has been used in the past to satisfy over 120 other product design constraints, illustrating the reusability and flexibility of the component.

Design Limitations

Using the Linear Programming format, we can now illustrate the design limits, as follows:

Design optimisation = $c + s$

The optimisation function is subject to the following limitations (A,B,C from Table 1 above):

(A)= $6c + 3s \geq 120$ Product Line/Family Constraints

(B)= $8c + 4s \geq 100$ Product Line/Family Solutions

(C) = $5c + 2s \leq 200$ Product Line/Family Constraints

The following represent the overriding fixed/explicit constraints originally set for the new software product.

(D)= $c \geq 10$ Explicit Constraints Satisfied (*)

(E)= $s \geq 8$ Explicit Solutions Satisfied (*)

(F)= $s \leq 65$ Product Line/Family Solutions to be Considered (*)

The linearity of design constraints and design solutions can now be exemplified by representing one dimension as zero, and the other dimensions coefficient as the denominator of the limitation factor. This is done for the reuse components (A), (B), & (C) as follows:

Reuse component A Where $\epsilon = 0$ then $s = 40$ (i.e. $120/3$)
 Where $s = 0$ then $\epsilon = 20$ (i.e. $120/6$)

This carried-out for reuse components 2 & 3 gives:

Reuse component B $\epsilon = 12.5$ & $s = 25$

Reuse component C $\epsilon = 40$ & $s = 100$

The other constraints (D), (E), & (F) are all fixed to one dimension. All of these values are then plotted on the graph to identify the feasible solution region (see diagram in section 3.2).