

# Lessons from industrial design for software engineering through constraints identification, solution space optimisation and reuse.

Denis Besnard

Centre for Software Reliability  
Department of Computing Science  
University of Newcastle upon Tyne  
Newcastle upon Tyne NE1 7RU, UK  
tel. 00 +44 (0) 191 222 8058  
Denis.Besnard@ncl.ac.uk

Anthony T. Lawrie

Centre for Software Reliability  
Department of Computing Science  
University of Newcastle upon Tyne  
Newcastle upon Tyne NE1 7RU, UK  
tel. 00 +44 (0) 191 222 6858  
A.T.Lawrie@ncl.ac.uk

## ABSTRACT

Design is a complex activity that can be analysed from a wide variety of perspectives. This paper attempts to look at the individual 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.

## Categories and Subject Descriptors

### General Terms

Design; Human Factors

### Keywords

Industrial design, software engineering, cognitive psychology, human-error, design faults.

## 1 INTRODUCTION

Due to the inherent complexity of industrial-scale software engineering, the development of software artefacts usually relies upon collaboration of cognitive skills that can only be provided through group effort [1]. Furthermore, with the ongoing proliferation of information and communications technology, the potential for greater collaboration has become even more prevalent with the increase of distributed development teams [2] and the emergence of Open Source Software development communities [3].

Despite these trends, both long-established literature [4] and contemporary industrial research [5] reveals that a significant part of software development work remains subsumed in individual cognitive activity. Moreover, although many useful analysis and design methods/techniques have been pioneered in software

engineering's relatively short history (see [6, 7, 8]), the inherent conceptual nature of software [9] ensures that its construction continues to rely upon a 'craft-based' element of individual creativity, experience, and skill [10]. However, this over-reliance on individual human capabilities frequently results in system failure(s) caused through discrete human-design faults made during the development and/or maintenance phases of computer-based systems [11]. Whilst these characteristics set software engineering apart from classical engineering, it has also been argued that important lessons can still be learned from more mature disciplines [12, 13]. This is the philosophy of our paper also.

In respect to these concerns and characteristics of software engineering, this paper focuses attention upon the role of individual cognition and its potential for optimizing or degrading the design of software in computer-based systems. This involves drawing upon well established psychological research to emphasize a) the importance of constraints identification, b) its resulting influence upon solution finding and c) knowledge reuse during the design phase. These will be the three main areas covered in this paper. 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 [8, 10] and analysed insightfully from a cognitive perspective [14, 15, 16]. Here, it mainly consists in discovering a solution that addresses a design objective, namely, by identifying the constraints imposed by both explicit requirements and knowledge assumptions, seeking options in a solution space, and investigating the possibilities for reuse of previous solutions.

From a cognitive standpoint, design implies achieving a potentially fuzzy goal [17] that admits a variety of solutions [18]. Under this angle, designing implies finding paths in a solution space<sup>1</sup>, the latter being defined as the total number of possible intermediate steps that exist between the statement of the problem and its solution [19, 20]. Solving a design problem is particular in the sense that the solution space can be enormous, however some solutions to a given problem may already exist. So there is justifiably great interest in attempting solution reuse, for cost or time reasons [21].

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

---

<sup>1</sup> Psychologists usually prefer the term *search space*.

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 propose a three dimensional framework. Obviously, because of our individualistic psychological view of design, we do not consider other important influences (e.g. process feedback, organisational; see respectively [22, 23]).

### 3 A THREE-DIMENSIONAL APPROACH TO DESIGN

We wish to investigate three areas which we think can shed some light on the design process. Thus, the next three sub-sections will namely address a) constraints identification, b) optimisation of the solution space and c) solution reuse.

In our view, the complete set of design constraints involves identification of both explicit constraints (i.e. user/customer imposed requirements) and implicit constraints (i.e. the designer's technical interpretation of these explicit constraints). However, it is important that the reader is aware that our interpretation and usage of the term 'constraints' is purely technical, in nature. It relates solely to functional attributes of the artefact and not to non-functional attributes (such as budget, schedule, tools used etc.; important though they still are [24]).

#### 3.1 The Identification of the Explicit Constraints Imposed by Requirements

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 more concrete view of these design conceptions, we will now consider an engineering example: the Tacoma Narrows bridge, in the USA.

##### *The Tacoma Narrows bridge<sup>2</sup>*

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, in the context of the Narrows. It caused the designer to consider the deflection theory as a viable design option.

---

<sup>2</sup> Unless otherwise noted, the source for the material exposed in this section is in [13].

The best-known cause of human cognition failure is the complexity of a problem [25]. 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 [26] or electronics operators [27]. Disregarding data is relevant to the design phase also. Bonnardel and Summer [28] 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* [29]) that does not accurately reflect the actual problem [30, 26]. 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<sup>3</sup>. 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 linked to the solution space. This position will be framed using the graphical version of the linear programming technique. It will be used to highlight the 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 [31]) and often included also in management decision-making literature [32]. 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. Moreover, we do not wish to imply that software design decision-making is strictly linear: the highly conceptual nature of software development has long been considered as containing non-linear characteristics [9]. Thus we do not assume that the design process can be reduced to a linear mechanism. Instead we use the graphical version of the LP technique as an *iconic model* ([33] quoted by [24]) to represent visually the role of constraints identification in optimising the solution space. 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 [31] (chapters 17-20) for a more complete coverage of using the LP technique.

Consider the following scenario:

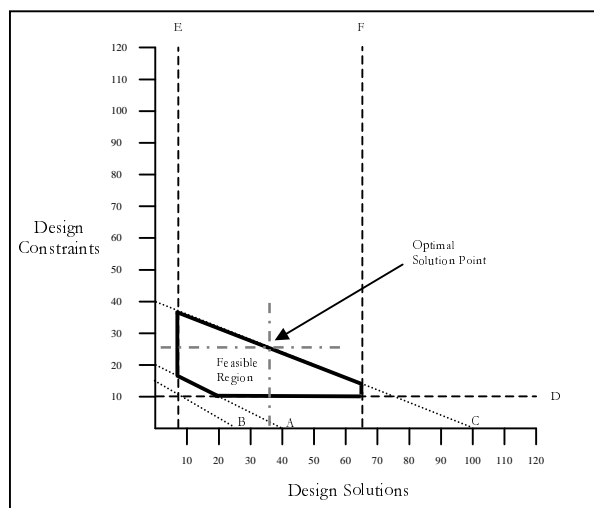
*A software company that manages its software development operations along product lines and product families to maximize software component and source code reuse. A software engineer, as design authority, is assigned to lead the technical development of the new software product. This new product requires 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.*

---

<sup>3</sup> Erroneously widening or narrowing the solution space can be caused by commission and omission errors, respectively (see section 5).

Because this scenario involves the optimisation of only two decision variables (i.e. design constraints and potential reuse solutions) the graphical method of LP 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 to search in.

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 (see Figure 3, section 5). 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 solutions<sup>4</sup>.



**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. If valid design constraints are overlooked, this will erroneously widen the solution space. It may then lead to the adoption of unsuitable design reuse solutions that later fail during operation when the effects of critical constraint omissions are experienced. Equally, admitting redundant<sup>5</sup> constraints will preclude viable potential solutions for reuse or adaptation. This erroneous narrowing of the solution space could result in unnecessary reinvention (and its consequent

<sup>4</sup> Please note that this technique is simplified here because none of the decision variables involved contain coefficient factors.

<sup>5</sup> The meaning of redundancy used here is the classical definition of an unnecessary function/object, and not the usage from the dependability community, meaning necessary function/object for additional support and strengthening of a system [34].

increased costs). It can even become a barrier to innovation through the designer(s) believing that the design problem set is unsolvable<sup>6</sup>.

The LP approach allows us to discuss the first stage in design optimisation by demonstrating how disregarding or adding explicit constraints affects the solution space<sup>7</sup>. This visual example aids conceptualisation of the intuitive notion of the optimal point between design constraints and solution options. It allows the designer to be aware of this important trade-off and compare several components against each other during the process of solution reuse.

We have now exposed how the aim of optimising the solution space could be established as a meta-design goal in its own right, for designers. The next section will document and discuss the potential catastrophic effects of deriving sub-optimal solution regions to search in 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 becoming more and more common practice<sup>8</sup>. Among other things, it is aimed at reducing costs and development time [36]. McClure [37] 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 [8]. Rather, the decision for reuse is supported by a compromise between the adaptability of the desired artefact, the cost of restructuring it [38] 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 [6]. 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 [4]). 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.

#### *The Ariane 5 failure*<sup>9</sup>

On the 4<sup>th</sup> 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

<sup>6</sup> In practice, relaxing some of the constraints in order to voluntarily widen the solution space is always a distinct possibility. In this respect, we only take a snapshot and preclude temporal change considerations.

<sup>7</sup> Although we have solely adopted LP as a way to graphically represent our theoretical position, some similarities can be found with other models, e.g. dealing with the negotiation of requirements (see for instance the Win Win model by Boehm *et al.* [35]).

<sup>8</sup> 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.

<sup>9</sup> Unless otherwise noted, the source for the material in this section is in [39].

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 faults identified resulted from a) some modules from Ariane 4 not being verified 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.

Cognitive psychology analyses reuse from the point of view of reasoning by analogy [18]. 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) [40, 41, 42]. One of the causes of errors lies in the possibility of not identifying important data in the source problem [43] or neglecting some discrepancies between the two problems' structures. The quality of the schema (see section 3.1) used by the designer is of relevance here. It may trigger itself in sub-optimal conditions, causing the analogical reasoning to be 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 latent<sup>10</sup> reuse fault [23, 34]. In the case of Ariane 4, the horizontal velocity values could virtually not go beyond the limits present in the software. So, during the development of the inertial reference system it was decided that protecting it from being made inoperative by excessive horizontal velocity values was not necessary<sup>11</sup>. 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.

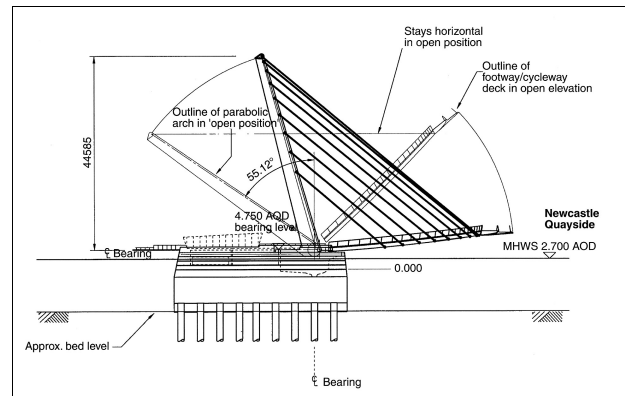
##### *The Millennium bridge, Newcastle upon Tyne, UK<sup>12</sup>*

As is often the case in design, the challenge was finding a novel solution that would compromise optimally all the explicit constraints. This is recognised as 'innovative design' [10]. 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 navigation channel;
4. preclude any construction on the quays themselves;
5. show some novelty.

In terms of explicit 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 crossing that would be suspended by another curve. Then tilting them altogether like the visor of a helmet or a blinking eyelid would create enough head room for bigger boats to pass (Figure 2).



**Figure 2: The Millennium bridge. Technical diagram (Clark & Eyre [44]).**

Although it is a suitable solution for most bridges, Clarke and Eyre [44] 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 be forced to violate some explicit constraints.

##### *The 32-bit memory architecture of the Eagle computer<sup>13</sup>*

The design of the 32-bit Eagle architecture (in the early 1980s) necessitated trading-off between constraints and reusable solutions. In this case, a single designer had to design a pioneering 32-bit memory architecture and provide time-sharing security protection. Thus, the design had to integrate these two main explicit constraints, narrowing the solution space dramatically. One existing reuse solution (which would have violated explicit constraints) 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 not only optimised explicit functional constraints but also non functional aspects, as the architecture also proved to be a simpler, cheaper, more efficient and more reliable system than any design previously. Again, in Tekinerdogan's view [10], this is innovative design.

These design successes above now allow us to discuss the second stage (implicit constraints) in design optimisation. 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 allow the determination of an optimal solution. In this respect, two different designers designing two different types of artefacts faced the same inevitable design

<sup>10</sup> Computing scientists will prefer the term *dormant* fault.

<sup>11</sup> Although the consequences of this decision were not fully understood.

<sup>12</sup> Unless otherwise noted, the source for the material in this section is in [44].

<sup>13</sup> Unless otherwise noted, the source for the material exposed in this section is in [45].

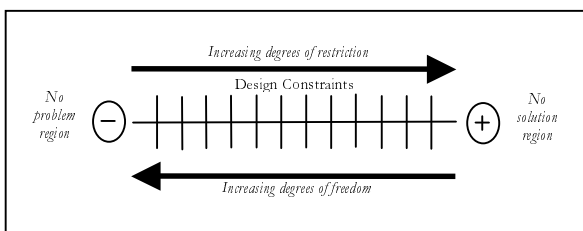
challenge. In each case they needed to critically explore the feasible region guided by the explicit constraints imposed by user/customer requirements. However, more importantly, they were able to remove implicit constraints through questioning the possible validity or adaptability of existing engineering practice and design knowledge. In the case of the Millennium Bridge this was going beyond the design assumption that all bridge crossings 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 optimised the solution space by widening it to include 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.

## 5 GENERAL DISCUSSION

Having considered comparable design success and failure cases using our 3-dimensional view, we can now define what we mean by an optimal solution space and its implications for both dependable and innovative engineering design:

1. At the explicit constraint level, this is when there exists no error of omission in the constraint identification process i.e. all valid explicit constraints are fully employed in defining the solution space to be searched.
2. At the implicit constraint level, this is when there exist no error of commission (i.e. intrusion of superfluous cognitive design paradigms) which would result in the persistence of redundant constraints that restrict the act of searching for an optimal solution. When this is achieved, as shown in Figure 3, it provides the maximal degrees of freedom and minimal degrees of restriction in finding an innovative design solution.

It is believed that design is progressively redirected and refined by integrating constraints imposed by a design option. As illustrated below, we think that constraints are spread over a continuum. The more constraints there are, the less possible solutions there are, meaning the more you have to trade-off with exiting a 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 a problem is defined by the constraints themselves.



**Figure 3: Constraints continuum**

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, design error(s) 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 an iconic linear programming representation. This conception graphically represented the area

where possible solutions exist (i.e. feasible region) and what the 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. Classically, it 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 often 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 problem is a mistake for it scales down comprehension of design to what the component to reuse can offer. As already shown in the cases provided, an erroneous 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 [46], political, economic, management and cultural influences [47] 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 elicited and understood to begin with. Yet, because of the highly complex and intangible nature of software, the elicitation of requirements has proved to be an 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 goal 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, ensuring that functional optimisation guides the design process. The other is reuse driven, which is directed by non-functional aspects (costs, schedule, etc.). 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 or introduce some constraints in the design process. A solution to this form of human error could then be extremely valuable in helping to prevent computer-related failures and thereby further improve the dependability of computer-based systems through increased error avoidance [48, 49, 34].

## 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. 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 Prof. Cliff Jones for his influencing initial design comments that helped motivate the writing of this paper. The authors also wish to thank Budi Arief, Jim Armstrong, Cristina Gacek and anonymous reviewers for useful comments; and the sponsor EPSRC for funding this research.

## 10 REFERENCES

- [1] Sallis, P., Tate, G. & MacDonell, S. (1995). *Software Engineering: Practice, Management, Improvement*. Addison-Wesley, Sydney, Australia.
- [2] Herbsleb, J. D., Finholt, T. A., & Grinter, R. E. (2001). An Empirical Study of Global Software Development: Distance and Speed. *Proceedings of the 23<sup>rd</sup> International Conference of Software Engineering*, Toronto, Canada, 12-19<sup>th</sup> May 2001 (pp. 81-90).
- [3] Raymond, E. S. (1999) *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, California, USA.
- [4] Weinberg, G. M. (1971). *The psychology of computer programming*. Van Nostrand Reinhold, London.
- [5] Robillard, N. & Robillard, P. (2000). Types of collaborative work in software engineering. *The Journal of Systems and Software*, 53, 219-224.
- [6] Pressman, R. S. (1992). *Software engineering. A practitioner's approach*. McGraw-Hill, London.
- [7] Bell, D. (2000). *Software engineering: A programming approach*. 3rd edition. Addison-Wesley, U.K.
- [8] Sommerville, I. (2001). *Software engineering*. Sixth edition. Addison-Wesley, Wokingham, UK.
- [9] Brooks, F. P. (1995). *The mythical man month: Essays on software engineering*. Anniversary Edition. Addison-Wesley, New York, NY.
- [10] Tekinerdogan, B. (2000). *Synthesis-Based Software Architecture Design*, PhD thesis, Dept. of Computer Science, University of Twente, The Netherlands.
- [11] Health & Safety Committee (1998). *The use of computers in safety-critical applications*. HMSO, UK.
- [12] Leveson, N. (1994). High pressure steam engines and computer software. *IEEE Computer*, 27, 65-73.
- [13] Holloway, C. M. (1999). From bridges and rockets. Lessons for software systems. *Proceedings of the 17<sup>th</sup> International System Safety Conference*, August 1999 (pp. 598-608).
- [14] Brooks, R. (1999). Towards a theory of the cognitive processes in computer programming. *International Journal in Human-Computer Studies*, 51, 197-211.
- [15] Von Maryhauser, A. & Vans, A., M. (1995). Program comprehension during software maintenance and evolution. *Computer*, 28, 44-55.
- [16] 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 5<sup>th</sup> Safety-critical Systems Symposium, Brighton, UK (pp. 111-120).
- [17] 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).
- [18] Burkhardt, J. M. & D tienne, F. (1994). La r utilisation en g nie logiciel: une d finition d'un cadre de recherche en ergonomie cognitive. In proceedings of *ERGO IA 94*, Biarritz, France (pp. 83-95).
- [19] Cordier, F., Denhi re, G., George, C., Cr pault, J., Hoc, J.-M., Richard, J.-F. (1990). Connaissances et repr sentations. in J.-F. Richard, C. Bonnet & R. Ghiglione: *Trait  de psychologie cognitive 2*. Bordas, Paris (pp. 35-102).
- [20] Newell, A. & Simon, H., A. (1972). *Human problem solving*. Englewood Cliffs, N.J., Prentice Hall.
- [21] McDermid, J. (1991). *Software engineer's reference book*. Butterworth-Heinemann, Oxford.
- [22] 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).
- [23] Reason, J. (1995). *Managing the risks of organisational accidents*. Aldershot, Ashgate.
- [24] Jackson, M. (2001). *Problem frames*. Addison Welsey, London, UK.
- [25] Amalberti, R. (1996). *La conduite de syst mes   risques*. Presses Universitaires de France, Paris.
- [26] Schanteau, J. (1992). How much information does an expert use? Is it relevant? *Acta Psychologica*, 51, 75-86.
- [27] 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).
- [28] Bonnardel, N. & Summer, T. (1996). Supporting evaluation in design. *Acta Psychologica*, 91, 221-244.
- [29] Reason, J. (1990). *Human error*. Cambridge University Press, Cambridge.
- [30] 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.
- [31] Lucey, T. (1992). *Quantitative techniques*. DP Publications, UK.
- [32] Cooke, S. & Slack, N. (1991). *Making management decisions*. Prentice-Hall, UK.
- [33] Ackoff, R. L. (1962). *Scientific method: Optimizing applied research decisions*. Wiley.
- [34] Randell, B. (2000). Facing up to faults. *The Computer Journal*, 43, 95-106.

[35] Boehm, B., Egyed, A., Kwan, J., Port, D., Shah, A. & Madachy, R. (1998). Using the Win Win spiral model: A case study. *IEEE Computer*, 31, 33-44.

[36] Richards, D. (2000). The reuse of knowledge: a user-centred approach. *International Journal of Human-Computer Studies*, 52, 553-579.

[37] McClure, C. (1992). *The three Rs of software automation. Re-engineering, repository, reusability*. Prentice Hall, Englewoods Cliffs, NJ.

[38] 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).

[39] Lions, J. L. (1996). *Ariane 5 Flight 501 failure*. Report by the enquiry board.  
http://www.cs.berkeley.edu/~demmel/ma221/ariane5rep.html

[40] Catrambone, R. & Holyoak, K. J. (1989). Overcoming contextual limitations on problem-solving transfer. *Journal of Experimental Psychology: Learning, memory and Cognition*, 15, 1147-1156.

[41] Novick, L. R. & Holyoak, K. J. (1991). Mathematical problem solving by analogy. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 17, 338-415.

[42] 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.

[43] Novick, L. R. (1988). Analogical transfer, problem similarity and expertise. *Journal of Experimental Psychology: Learning, Memory and Cognition*, 14, 510-520.

[44] Clark, G. M. & Eyre, J. (2001). The Gateshead Millennium bridge. *The Structural Engineer*, 79, 30-35.

[45] Kidder, T. (1981). *The Soul of a New Machine*. Back Bay Books, USA.

[46] Carroll, J. (1997). Human computer interaction: psychology as a science of design. *International Journal of Human-Computer Studies*. 46, 501-522.

[47] 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.

[48] Laprie, J.-C. (1992). *Dependability: Basic Concepts and Terminology*. Springer-Verlag Wien, New York.

[49] Neumann, P. G. (1995). *Computer-related risks*. Addison-Wesley, New York, NY.

## APPENDIX

### A.1 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. 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.

### A.2 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):

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

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

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

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

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

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

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

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

$$\text{Where } c = 0 \text{ then } s = 40 \text{ (i.e. } 120/3)$$

$$\text{Where } s = 0 \text{ then } c = 20 \text{ (i.e. } 120/6)$$

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

$$\text{Reuse component B } c = 12.5 \text{ \& } s = 25$$

$$\text{Reuse component C } c = 40 \text{ \& } s = 100$$

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.).