Goal-Diversity in the Design of Dependable Computer-Based Systems

A. T. Lawrie and C. B. Jones
Department of Computing Science
University of Newcastle University NE1 7RU
February 2002

Abstract

This paper sets out an argument for experimenting with some aspects of the "Open Source" development style in the creation of "Computer-Based Systems". The particular objective is to find ways of increasing the "Dependability" of systems. The most interesting facet of Open Source development in this connection is the use of multiple developers to introduce diversity into the design process. In addition to relying on the fact that no two developers are identical, the suggestion here is that asking each developer to emphasize different goals might result in solutions whose comparison and combination could increase dependability.

Keywords

COMPUTER-BASED SYSTEM, DEPENDABILITY, DESIGN PROCESS, DIVERSITY, FUNCTIONAL GOALS, GOAL-DIVERSITY, HUMAN-DIVERSITY, HUMAN-REDUNDANCY, NON-FUNCTIONAL GOALS, OPEN-SOURCE SOFTWARE PROCESS, REDUNDANCY.

1. Pre-Prologue

In the spirit of the proposed approach to development, the two authors (ATL, CBJ) have discussed many ways of presenting the ideas in this paper. In the end, two rather different styles are given below: in the Section titled "Prologue" (CBJ), an outline argument is presented; Sections 3 – 6 (ATL) expand on this argument and provide definitions of terms etc.

2. Prologue

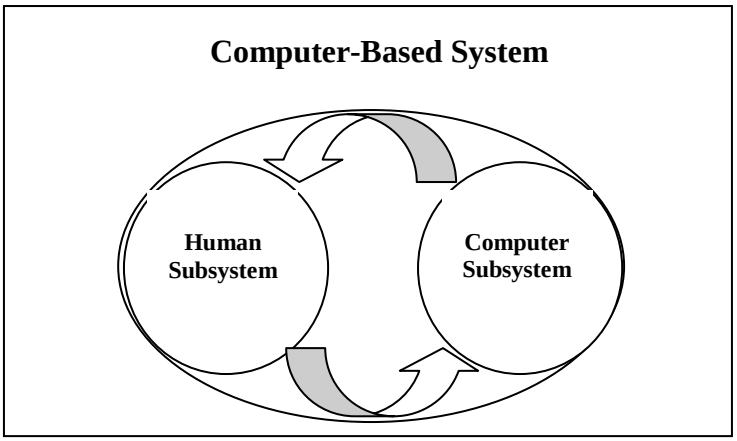
- The systems of interest are referred to as "Computer-Based Systems" to reflect the key role of people in the overall system.
- The aim is the creation of systems that are dependable.
- Dependability is often achieved by careful use of redundancy.
- Systems are created by (groups of) humans.

- The interest then is to use redundancy in the design and development process.
- Such redundancy is present in projects described as “Open Source” developments.
- Humans inevitably introduce an element of diversity into any task.
- Setting different secondary goals for designs can enhance diversity.
- Careful comparison and combination of diverse solutions could result in dependable systems.
- The whole discussion can be related to standard notions such as the “attributes”, “impairments”, and “means” of dependability.

3. The Design Process of Computer-Based Systems

The phrase “Computer-Based System” (CBS) is used in a socio-technical sense that extends the chosen view of systems to include both humans and computers. This is illustrated in Fig 3.1.

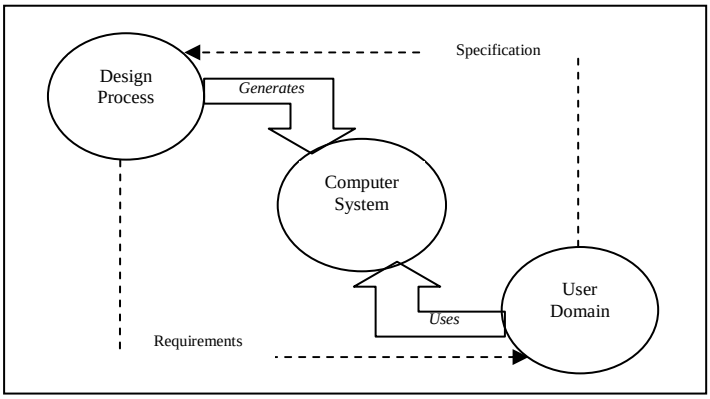
Figure 3.1 **Two-Subsystem CBS Definition**



This definition has ramifications since it interprets the human subsystem mainly in terms of the user-domain and emphasises the need to balance and consider the influences of both technology and social issues in their development [Mason & Willcocks, 1994]. Although this is critically important, it ignores the other key human role in the subsystem that concerns the design and development of the computer system.

To clarify this point, it is possible to illustrate how the view of a CBS could be extended to include three subsystems:[1] 1) the design-process, 2) the computer system, and 3) the user-domain. This view is illustrated in Fig 3.2.

Figure 3.2. **Three Subsystem CBS Definition**



When viewed in this way, a number of important considerations can be discussed regarding the design and use of CBS.

1. The two human subsystems can be considered as natural-systems and the computer-system can be considered as an artificial system
    - Natural systems can set and change their goals whereas artificial systems are designed by humans to fulfil human needs and therefore have their goals designed into their function[2].
2. Here then, we can appreciate the important dominating relationship between the design process and the computer-system. The design process generates the computer-system (see [Jones, 2002]).
3. The goals of the design-process will therefore have an overriding influence upon the eventual computer-system produced. Any errors or shortcomings that arise in the design process can manifest themselves as defects or deficiencies in the computer-system (see Section 4)

[1] It should be noted that the terms for design process and end-user domain include both their primary secondary environments (see: [Cooke & Slacke, 1991]). Additionally, the term computer-system includes both hardware and software components.

[2] Whether natural or artificial, most systems are goal-directed i.e. their behaviour is purposive towards achieving and maintaining some desired state [Heylighen, 2001].

4. The design specification represents a formal interpretation of the user-domains requirements, in terms of goals of the computer-system: misunderstandings or omissions will result in a computer-system that does not fully satisfy this purpose.
5. Over time, the user-domain may change its goals. Unless the design-process can change the computer system's "designed-in" goals accordingly, the computer-system will no longer fully satisfy the user-domains goals.[3]

3.1 Section Summary

The key point from this Section is that the design process is a key (socio-technical) subsystem in a CBS. Its players are responsible for the interpretation, creation, and evolution of the user-domains goals and expectations – via computer system development. Whilst traditional software engineering has improved the technical approaches to CBS development, there are criticisms that there has been little progress into understanding and progressing the socio-technical influences upon the software development process [Beynon-Davies, 1999]. Furthermore, as indicated in this Section, the existing CBS definition, at best, makes its inclusion implicit.

4. Dependability of Computer-Based Systems

The stance taken by the dependability community is to accept that, in any non-trivial software system, it is almost certain residual design faults will remain in the CBS [Randell, 2000]. Therefore, dependability is concerned with how, at the system level, such systems can be designed and developed to provide an acceptable continuity of service in the event of faults giving rise to errors that may affect the expected delivery of service (see Section 4.1).

In order to help achieve this goal, a large body of theoretical knowledge and technical application has been combined into a conceptual framework. At the highest level, this framework identifies three principal factors influencing dependability.[4]

[3] Software evolution and maintenance is problematic and extremely costly. It may consume 80% of a software system's total life-cycle costs. Legacy systems represent computer-systems that can no longer fully satisfy the user-domains goals [Sommerville, 2001].

[4] Unless otherwise cited, Sections 4.1 through to 4.3 is with direct reference to [Laprie, 1992].

4.1. Impairments to Achieving Dependability

The impairments of dependability are concerned with the nature of problems in complex systems. These are faults, errors, and failures:

- **Faults:** are the hypothesized cause(s) of a system error. A fault becomes active when it produces an error
- **Errors:** are any part(s) of the system state that is liable to lead to a subsequent system failure. During system execution, the presence of active faults can only be determined by the detection of errors.
- **Failures:** occur whenever the delivered service no longer complies with the specification - this being an agreed description of the system's expected function and/or service.

4.2. Means to Achieving Dependability

There exists a collection of methods and techniques to promote the ability to deliver a service on which reliance can be placed, and to establish confidence in the system's ability to help accomplish this:

- **Fault prevention:** how to prevent fault occurrence or introduction into the CBS system.
- **Fault removal:** how to reduce the presence (number, or seriousness) of faults;
- **Fault tolerance**: how to provide a service complying with the specification in spite of faults;
- **Fault forecasting:** how to estimate the present number, the future incidence, and the consequences of faults.

4.3. Attributes to Achieving Dependability

System properties can be identified that help reveal certain desirable attributes of dependability. However, depending upon the users and application domain, such properties may be more (or less) emphasized. The main attributes are [Laprie, 1995]. :

- **Availability**: readiness for usage;

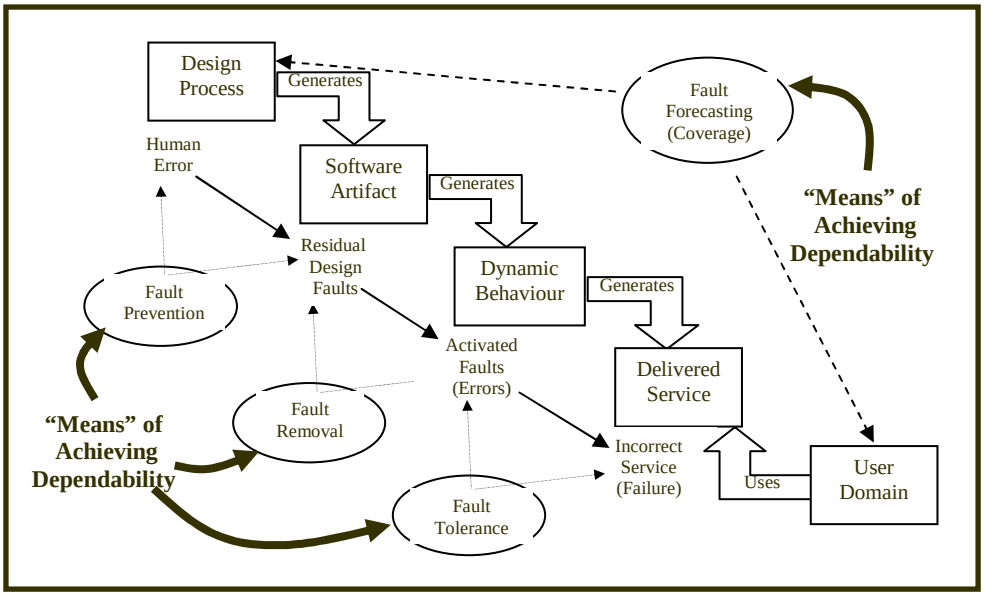
- **Confidentiality:** non-occurrence of unauthorized disclosure of information
- **Integrity:** non-occurrence of improper alterations of information
- **Maintainability:** the ability to undergo repairs and evolution
- **Reliability:** continuity of service;
- **Safety:** non-occurrence of catastrophic consequences on the environment

Furthermore, some of these may be compound attributes generated from other ones. For example, **Security** is seen as being the combination of attributes **Integrity, Availability**, and **Confidentiality.**

4.4. A CBS View of Achieving Dependability

By mapping these *means* and *impairments* to an extended version of the three subsystem CBS definition (cf. Fig 4.1), the dependability factors documented above can be illustrated to reveal the generic strategies available to achieving greater dependability of CBS during their design and development.

Figure 4.1 **Main CBS Dependability Strategies**



Firstly, as already highlighted in Section 3, Fig. 4.1 shows the dominating influence of the design process. Any human errors or oversights in the design process can quickly result in generating design faults in the software artefact[5], which then, during execution, become activated into errors and result in service delivery failures later during operational use. Secondly, the main dependability strategies employed are also shown, in terms of *fault prevention* in the design process, *fault removal* in the software artefact, and *fault tolerance* to intervene and limit errors causing service failure during operational usage.

Fault prevention, although an important *means* to achieving dependability, is seen as a *'general'* system and software engineering responsibility, while *fault-tolerance* is seen as a specialist area concerned directly with achieving increased dependability at the system level. *Fault removal* may be employed by both fields – either at the process level (i.e. testing) or system level (i.e. fault-masking) (see: [Laprie, 1992], [Randell, 2000]).

Fault forecasting is concerned with techniques (i.e. fault-injection) to ensure a representational *"coverage"* of the system's intended operational usage. It is a responsibility of both general software engineering, in terms of gaining a thorough understanding of the user domain to support *fault-prevention* strategies (i.e. specification completeness) and a specific *means* of achieving increased dependability that supports *fault-tolerance* – regarding the anticipation and generation of more accurate fault assumptions to decide which *fault-tolerant* mechanisms will be most effective to apply [Randell, 2000].

[5] The software artefact represents a "white-box" compositional view of the system…whereas the computational behaviour represents a "black-box" dynamic execution view of the system (cf. [Jones, 2002]). Of particular interest is that, since the user can only judge the provision of service from a black-box view of the CBS, perceptions of service failure may vary from one user to another. Equally, while faults and errors (in an absolute technical sense i.e. a fault manifests into an error which then affects the computational behaviour) may occur at the white-box and black box levels, it is only considered (i.e. judged) a failure if it becomes undesirably perceptible to the user's view of required service delivery. Therefore, a failure in the CBS can occur without a failure in the service (but not vice versa – hence the 'generates' relationship in diagram 4.1)

4.5 The Complexity Involved in Achieving Dependable CBS

Dependability is an inclusive concept but all of its aspects can have very subtle, interdependencies, interactions, and interpretations in any specific context. It can also be seen that the dependability attributes relate not to "what" functionality is delivered, but "how" that functionality is delivered. They therefore relate to the desirable non-functional qualities that promote CBS dependability (see: [Lamsweerde, 2001], [Jackson, 2001]). For example:

- **Reliability** of CBS is where the system behaviour *provides* (and only provides) that functionality needed for service delivery
- **Availability** of CBS is where *authorised access* to the required functionality can be provided whenever service delivery is needed.
- **Safety** of CBS is where the required functionality does not result in service delivery that can result in *damage* to the user or wider environment
- **Security** of CBS is where the required functionality does not result in service delivery that can be mitigated by *unauthorized* accidental or malicious *access* by others.
- **Maintainability** of CBS is where required service delivery *changes*, over time - by the user-domain, can be satisfied by equivalent functionality changes in the computer system.
- **Performability** of CBS is where the required functionality can be provided at a *time* needed to provide the delivered service. [6]

However, not only may these non-functional qualities vary from one specific application to another, but also, as [Bell, 2000] notes, in any design or development context the relationship between these non-functional attributes can be either complementary or conflicting (these relationships may however vary also between specific applications). A simplified and generic view is shown in Fig. 4.2.

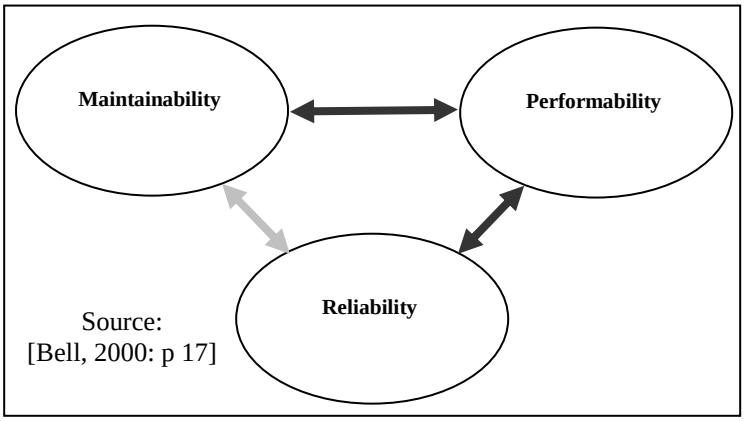
Here, we can see that while reliability and maintainability may often complement each other during design and development, trying to achieve increased performance, also, may well conflict with, and militate against fulfilment of, the other two design goals[7].

---

[6] The *"close proximity"* of non-functional considerations relating to timeliness of functionality provision and the actual functionality provided in designing and developing real-time applications is highly visible and usually made explicit.

[7] The pursuit of increased structural flexibility offered by low data coupling, information hiding, and high cohesion to achieve maintainability goals will often also complement the achievement of reliability goals through increased faults and errors control and containment offered by program scope localisation.

Figure 4.2: **Complementary and Conflicting Goals in CBS Design**



4.6 Redundancy and Diversity

One key general weapon to achieve Dependability is *Redundancy*. In many engineering applications, one can "over engineer" by, for example, introducing more strength than is required in materials or leaving more than minimum time. In situations where random failure or decay is the enemies of dependability, the use of multiple instances of a component (as in "Triple Modular Redundancy") can increase dependability. But design errors are not random and the execution of three copies of a flawed algorithm will do nothing to remove their inherent undependability. Redundancy can be utilised in software systems but it must be weeded to a way of achieving diverse solutions.

4.7 Section Summary

The key point from this Section is that, in complex CBS, residual design faults from the design process are almost inevitable. Awareness of this has given rise to a range of dependability strategies to prevent, remove, tolerate, and forecast faults, errors, and failures that may occur. To aid this goal, dependability employs a range of mechanisms that leverage redundancy and diversity in fault tolerant strategies. However despite the dominating role of the design process, concepts of redundancy and diversity are little

---

However, the pursuit of performance goals using such structures as inline-functions (c.f. Prata, 1995) and fast complex algorithms can often reduce program comprehension and produce unanticipated and undesirable side-effects – that can result in faults, errors, and failures later. Therefore undermining potentially both maintainability and/or reliability goals.

used to facilitate the other dependability strategies or overcome the inherent difficulties of promoting and assuring the desirable attributes that embody dependable CBS.

5. Human Redundancy and Human Diversity

5.1 Human Redundancy

It has been argued in Section 4 that redundancy is fundamental to improving the dependability of engineered systems. In particular, it is a core feature of the many fault-tolerant strategies that have been applied to improve CBS dependability. What is less well understood is how forms of human-redundancy can be applied to the other dependability strategies of fault-prevention, fault-removal and correction, fault-coverage – involving the design process. Furthermore, it is legitimate to consider how human redundancy may also help promote the integration and assurance of important non-functional attributes that reinforce the dependability of CBS.

5.2. Human Redundancy and Open Source

The emergence of Open-Source software development is a recent example of how human-redundancy can be employed in a highly decentralized way [Raymond, 1999]. Factors common (see [Gacek et al., 2001] for a discussion of different Open Source attributes) too many Open Source projects include development through geographically remote collaboration across the Internet. The main form of communication and coordination is usually via email, website domains, and central source-code repositories. Such products as the Linux operating system, the Kde desktop, and the Apache web-server have become highly successful with both industrial and domestic users and advocates.

The voluntary and indirectly subsidized nature of Open-Source development [Meyer, 2000] allows the potential for a level of duplication of development effort that could be rarely (if ever) matched and supported in traditional software development projects. Successful Open-Source projects can be supported by hundreds, and sometimes thousands, of contributing developers. Furthermore, [Yamouchi et al., 2000] indicates that a combination of voluntary effort and self-appointed work allocation supports the

potential for duplication of effort and overlapping development. Such human-redundancy characteristics are often found in high reliability organisations where dedicated duplication and overlapping responsibilities are employed for crosschecking and verification [Viller et al., 1997]. This is, however, in stark contrast to traditional software engineering practice where commercial constraints view human resources as economically limited and any duplication could result in increased costs and reduced market competitiveness. Consequently, traditional software development has increasingly been managed along concurrent engineering lines where project schedules are expedited by ensuring that the work is divided up and allocated on an individual task basis. [Weinberg, 1971] argued that this inhibits the ability to judge the quality of programs, as no source exists for generating comparative criteria.

5.3. Human Diversity

As noted in Section 4.6, software redundancy also requires diversity. Therefore, implicit in the discussion in Sections 5.1 and 5.2, is the fact that human redundancy is valuable because humans, at both the physiological and psychological levels, have a significant degree of variation ([Lewontin, 1982], [Kandola & Fullerton, 1998]). For example, at the cognitive level, humans have different intellectual abilities, experiences, knowledge, and personalities that have already been studied and considered beneficial for fault-detection and correction [cf. Westerman et al., 1997]. If this were not the case, human redundancy – in terms of task duplication and overlapping, would be of no value since the redundant human resources would always reach the same view and make the same mistakes[8]. Despite this, the levels of diversity required in achieving some fault-tolerant mechanisms (i.e. n-software channels for Triple Modular Redundancy) are very high and even with attempts to 'force' design diversity, within the task environment [Popov et al., 1999],

[8] It should be noted, however, that human redundancy does not necessarily provide the equivalent diversity expected. Such influences as organizational culture, management, and group/team influences of "group-think", "freeloading" and "dominating individuals and group-norms" can all mitigate and undermine the expected diversity benefits from redundant human resources ([Byrne, 1991], [Cooke & Slack, 1991]). To achieve expected levels of diversity, redundant resources, and the environment in which they interact, must be carefully managed (cf.[Kandola & Fullerton, 1998]). Therefore, "human diversity" must be assured at the social level before it can be benefited from at the technical level.

humans are still prone to common-mode-failure[9]. However, this paper is not concerned with such issues[10], but instead, focuses upon how human resources could be reorganized within the software development process to employ human redundancy and diversity to increase the potential of the other existing *means* of dependability - such as fault prevention, fault-removal, fault-coverage. Furthermore, how could the diverse usage of human resources improve design for dependability – in terms of assuring the integration of desirable dependability attributes (e.g. Security, Reliability etc)? Three areas are explored in Sections 5.3.1 to 5.3.3, which also draws upon the Open-Source development process where appropriate.

5.3.1 Increased Fault-Removal and Correction

One of the most visible benefits of the Open-Source development process is its increased bug finding ability through massive peer reviews of submitted source-code. This was characterised by [Raymond, 1999] in his seminal paper on evaluating the Open-Source software development approach as:

> *"Given enough eyeballs, all bugs are shallow." (p. 41)*

This informal approach to using human diversity for bug finding was, however, exemplified much earlier by [Weinberg, 1971] in his "egoless programming" philosophy.

To return to the main strategies of achieving dependability in CBS (see Fig. 4.1), this example demonstrates how the design process can improve the dependability of CBS through leveraging existing cognitive diversity [Westerman et al., 1997] for increased fault detection, removal, and correction before deployment.

---

[9] This is where two (or more) developers make the same human error during system development resulting in multiple version failure under the same functionality demands (cf. [Knight & Leveson, 1986]).

[10] Although we do not wish to imply that design and cognitive diversity for minimizing common-mode-failure is unimportant. Only that human redundancy and diversity may also be valuable in the other areas of promoting dependability e.g. fault-prevention, removal, coverage and assuring design for dependability.

5.3.2 Increased Fault-Coverage

A much less visible characteristic of Open-Source development is the belief that, because of the voluntary nature of Open-Source, developers naturally "gravitate" to software development work they are naturally interested and/or already knowledgeable in performing [Lang, 2000]. This choice-orientated resource allocation, combined with the reality that Open-Source developers are also users of the software they develop [Gacek et al., 2001] reveals that Open-Source developers have an intrinsically enhanced understanding and knowledge of the user-domain in which the software will be deployed. Consequently, this improves the particular developer's ability to anticipate potential usage exceptions, and generate accurate fault, error, and failure assumptions that the system may be susceptible too during operational usage (cf.[Randell, 2000]).

On a less positive note, the Open-Source development approach appears only to develop software where there are well-established existing systems to clone/improve upon, or where the development knowledge required is well exposed [Meyer, 2000]. This seems to limit major interest to the development of such systems as commercial-of-the-shelf applications (e.g. desktops, word-processors, office-applications, RDMS etc) or systems and systems development type software (e.g. compilers, operating systems, programming IDEs etc). Therefore, while it is possible that the Open-Source software development approach may be capable of developing such systems more dependably, it highlights that the Open-Source software process may be unworkable for development of software systems for specific application domains like business applications, process control software, or medical information system [Gacek et al., 2001]. It appears that, in contrast to traditional development approaches, Open-Source development is not subject to the additional requirements engineering challenges with which traditional software engineering is often faced.

However, influenced by the Open-Source phenomenon, [Anderson, 1999] carried out an experiment in massive parallel requirements engineering to derive a security specification for a (notional) national lottery system. He reported positively on how human diversity can be used to increase the reliability of computer-based systems through enhanced

specification completeness and consistency checking. His findings are consistent with social constructivist views that different confirming observations that support each other increase the reliability of what is perceived [Heylighen, 2001].

While the Open-Source process indicates that greater fault-coverage can only be achieved though human redundancy and diversity in known or well-established user-domains, [Anderson, 1999]'s example begins to indicate otherwise. It suggests that such an approach could be utilized as a fault-prevention approach to decrease the risks of errors of omission possibly resulting in functionally deficient computer systems through specification incompleteness due to ignoring or omitting important user-domain details in deriving CBS requirements.

5.3.3 Increased Problem-Solving and Solution Finding

A much more ambiguous possibility, in Open-Source development, is the increased potential for problem solving and solution finding through massive forms of human redundancy and diversity. In his review of formal and informal design philosophies [McPhee, 1997] noted that the benefits of the informal route was the increased learning and specific knowledge acquisition that an explorative and iterative approach to design offered. Such views are reinforced by the experiences of [Raymond, 1999] when he designed the Open-Source product "Fetchmail". During the project he became convinced that having many developers scrutinize source-code designs can result in someone helping you reframe the problem and simplify the design solution. He noted that:

> *"…It is not only debugging that is parallelizable; development and (to a perhaps surprising extent) exploration of design space is, too…at a higher level of design, it can be very valuable to have the thinking of many co-developers random-walking through the design-space near your product…exploration essentially by diffusion…This works very well. "* (p. 47-52).

Software design and development has long been recognized as one of the most complex tasks imaginable [Brooks, 1995], [Glass, 1998]. This is because a designer is often faced

with a multiplicity of design goals to accomplish [Weinberg, 1971]. In such situations, studies have shown that, as a result of this complexity, developers will focus on the most prioritized goals through treating other important system attributes as "free-variables" to be traded-off in their achievement [Weinberg and Schulman, 1974]. However, it is suggested here that the Open-Source software process is 'geared' towards openly resolving such technical conflicts. A justification for such a view is that Open-Source software development is driven by, and relies upon, experienced and talented developers having ongoing "self-interest" to continue supporting the product [Sanders, 1998]. To achieve this, the Open-Source approach must be focused upon searching and finding appropriate design solutions to resolve such conflicts in order to accommodate the interests of the developer majority[11]. A good example of this is discussed in [Pettit et al., 2001]. They argue that the reason why the Linux kernel was designed to support dynamically linked modules was through a very large group of people pursuing their own individual interests of wanting to add or remove large sections of functionality in a convenient manner to suite their own technical and functionality needs. In short, they believed the stimulus for such a solution was because:

> *"Linux was made by a very large group of people with a very diverse set of objectives." (p. 40).*

Although this is a much less certain attribute of Open-Source software development, there are indications that a combination of massive human-redundancy and diversity of individual development goals helps to overcome the inherent complexity of software development by supporting increased problem reframing and solution finding. This is achieved not only through increased design ideas generated, but also through greater visibility[12] of the many design conflicts and trade-offs regarding important system attributes that developers make during software development. To return to the theme of

[11] The Apache "shared-leadership" project is a good example of how design is the result of accommodating and resolving design solutions through majority consensus [Fielding, 1999]. Furthermore, it is suggested that the inability to resolve technical design conflicts can often result in "code-forking" of one project into two different design directions (cf. [Moody, 2001]).

[12] See also [Lamsweerde, 2001]: the review of "goal-orientated requirements" and the potential for goals to making requirements and design conflicts more identifiable.

Section 4, it is suggested that leveraging human redundancy and diversity in a similar way may improve the integration and assurance of such desirable non-functional attributes of dependable software systems in CBS design.

5.4 Section Summary

The key point of this section was to explore how human forms of redundancy and diversity may be employed within the software development process to promote the other dependability strategies of fault prevention, removal, coverage, and design for dependability. It has been identified that human forms of redundancy are relevant for CBS design as they are inherently differentiated and therefore bring a level of diversity required for software development. As examples, we have identified human redundancy and diversity issues raised by the recent phenomenon of Open Source Software development. The Open Source approach begins to suggest that human redundancy and diversity can be utilised favourably for all three dependability strategies. Furthermore, there are indications that human redundancy and diversity increase problem solving and solution finding through the leveraging of others design views/ideas and subsequent resolution of self-orientated development goals. It is suggested and discussed in Section 6 below that such an approach to software development may also help increase the integration and assurance of desirable dependability attributes during CBS creation.

6. Goal-Diversity

6.1 Engineering Human Diversity

As discussed in Section 3, the design process is a natural system that sets and changes its own goals. Furthermore, it has a dominant affect on the computer systems it creates. Since a computer systems are artificial systems that has its goals designed-in, the goals pursued by the developers will have an overriding influence on the eventual system produced. Any omissions, conflicts, or mismatches are likely to result in defects and deficiencies in the system it eventually generates. Therefore, one way to improve control and help reduce such problems is to influence the goals of the developers through careful

goal setting[13] of required design goals to be achieved and maintained throughout the development of CBS.

Taking the lead from existing approaches of "forced" diversity [Popov et al., 1999], cognitive-engineering [Westerman et al., 1997], along with goal-setting in management theory [Latham & Locke, 1979], diversity can be engineered through goal-diversity, also, to achieve purposive design behaviour by deliberately predisposing individual developers to pursue and maintain different goals[14]. Such an approach would contain both convergent and divergent influencing factors often considered vital for creating effective and successful "heterogeneous teams". As [Shepard, 1964] advised:

> *"Variety is the spice of life in a group, so long as there is a basic core of similarity."* [p. 118].

With "goal-diversity" the similarity core will be maintained by contributing developers all pursuing the same functional specification, whilst variety will be engineered-in at the technical task level through ensuring that each individual developer is responsible for promoting the influence of a different desired dependability attribute to be assured and integrated throughout the design/implementation phase of the project (e.g. contributor 1 = Security; contributor 2 = Reliability; contributor 3 = Maintainability, etc.). Furthermore, such convergence and divergence is congruent with uses of human redundancy in high-

[13] The value of goal setting has been long recognized since "Management by Objectives" was first advocated by management guru Peter Drucker in the 1950s for its ability to improve performance, create purposive goal-directed behaviour, and alter attitudes in both individuals and groups, see: [Latham & Locke, 1979] i.e. management by objectives, [Demarco & Lister, 1987] i.e. goal alignment and "shared-goals" in teams, [Covey, 1992] i.e. personal improvement. It should be noted, however, that such philosophies were orientated towards creating alignment and harmony. The usage of goal-setting advocated here is to deliberately create dissonance and disharmony. In this respect, it is a perversion of the purpose that goal-setting is normally utilized.

[14] It can be argued that approaches to "diversity" so far in software engineering have been "implicit" forms that rely on subconscious and pre-existing variation in humans. For example, design-diversity attempts to stimulate diversity only through the artifacts used in the task of software development (i.e. methods, tools, techniques). Cognitive diversity also only attempts to leverage subconscious diversity through appropriate psycho-metric measurement and subsequent team member selection. However, goal-diversity attempts to engineer and stimulate diversity at the conscious level of the task by "explicitly" affecting the approaches to the task - irrespective of the artifacts used and natural cognitive variation of the specific individuals involved. However, it would be interesting also to see the impact of explicit diversity approaches on the others and their combined potential for increasing human diversity.

reliability organisations mentioned in Section 5.2. (i.e.. duplicative and overlapping at the functional level).

The following Section from 6.2 to 6.4 discusses the envisaged benefits that may be derived from such an approach – along with initial ideas of how it could be implemented.

6.2. Goal-Promotion

Non-functional attributes of a software system are more likely to be fulfilled when they possess a close proximity to the required functionality. For instance, system performance, in terms of its timeliness of functionality, is so closely linked to required functionality in 'real-time' systems that this non-functional attribute will be explicitly promoted as an important development goal to be achieved during the design process. However, maintainability of the CBS is often so remote from present required functionality that it is more likely to remain and implicit user-domain expectation of the system in the future. When this happens achievement of the non-functional goal is ultimately left to the responsibility of the particular developer(s) involved – with regards to their discretion, skill, and professionalism. As we have seen, the complexity and multiplicity of the software design task will often result in such implicit goals being traded-off for more explicitly demanded ones. Consequently, the integrity of the CBS, in terms of its maintainability attribute, may well be compromised and will not become an issue until years later when the user-domain requires the system functionality to be changed, corrected, or enhanced. Therefore, if one wants a CBS to possess the desirable non-functional attributes that support dependability, such attributes must be explicitly promoted as desired non-functional goal(s) of the system during development.[15]

[15] The value of making explicit non-functional requirement qualities is well established in manufacturing and is known as a "design for X" approach. Here, for example, they may ensure the subsequent commercial effectiveness of the manufacturing process at the initial product development stage by making explicit the non-functional requirement of "design for ease of assembly" to reduce costs and speed-up production during manufacture (cf. [Chen, 1999]). It appears, from Bill Gates recent email (see: [Boutin, 2002]), that software producers like Microsoft, are now placing an increased value and priority on making explicit non-functional dependability attributes – such as *Availability* and *Security* in order to achieve and realise their design goal(s) in the future of ***"Trustworthy Computing"*** for .NET.

The idea of goal promotion is to achieve both increased diversity and coverage of important non-functional attributes, at the system level, of CBS by utilising human redundancy and diversity to implement the same functional specification as a primary goal, during development, while deliberately predisposing individual developers of the contributing group to take responsibility and ownership of an important non-functional attribute as their own secondary goal during implementation. This not only reduces the complexity of the task, in terms of removing the multiplicity of goals an individual developer must consider, but also ensures sufficient coverage of important desirable attributes of the CBS.

6.3. Iterative Design Phases

The notion of goal-promotion from Section 6.2 is an important prerequisite for this phase to work correctly - as it relies upon continued goal-ownership of secondary non-functional goals throughout the iterative stages of design review and goal relationship identification (see: Sections 6.3.1 – 6.3.2). This ensures that individual developers continue to defend their own secondary goals and critically evaluate and question the other developer's secondary goal design decisions throughout the design and development of the system. As already discussed in Section 5.3.3, it is proposed here that a number of iterative design and refinement phases (see Sections 6.3.1 – 6.3.2) will promote increased learning and problem domain understanding. This is necessary with such an approach as it was already noted in Section 4.5 how the desired non-functional attributes of dependability have subtle, interdependencies, interactions, and emphasis given any specific application. Therefore, because of this novelty, it will be necessary for the developers to acquire a deeper understanding before a specific set of evaluation criteria will emerge that will allow the developers to be able to judge the design rationales to assess more accurately whether a particular non-functional goal has been achieved or not (or to what degree it has been achieved or not).

It is also suggested here that a deeper understanding of the problem domain will also support other dependability strategies of increased fault-prevention and fault-coverage through both natural cognitive-diversity that exists in humans – but also because of the

increased sensitivity of developers to other developers design decisions and fault, error, failure assumptions as the design progresses. It is suggested that this will also allow them the increased ability to anticipate potential defects and deficiencies of the CBS in advance of its deployment [16](i.e. links with fault-coverage and fault-prevention).

6.3.1 Peer-Review

The benefits of peer review inspections of program design have already been discussed in the context of human redundancy and diversity and the Open-Source approach. The "many eyes" affect not only supports fault detection and removal strategies of dependability but, (as noted in Section 6.3 above) in the context of this approach, it also increases the visibility of the overall development through stimulating greater insight through reviewing diverse approaches to the design problem by other contributing developers.

As an initial attempt at envisaging how it should be undertaken, it is suggested that the beginning of the design phase should be undertaken by developers working in complete isolation from each other until they have a deeper overall knowledge of the design problem.[17] After an initial attempt at the design each developer then reviews all of the other developers' diverse attempts at the design problem. Along with promoting fault detection and removal strategies, the purpose of the review will be to identify important goal relationships that exist between the diverse designs. This is further explored and discussed in Section 6.3.2, below.

---

[16] In fact the approach advocated may also be beneficial for the fault-tolerant strategy - as improved problem domain understanding may also help in generating more accurate fault-assumptions and help in determining the most effective fault-tolerant mechanisms to employ (see: [Randell, 2000] for more on the importance of generating accurate fault-assumptions). .

[17] Although the subject of another paper and outside the scope here, it has already been highlighted that diversity can be mitigated by group, organizational, and management influences. The justification for this isolation is therefore to preclude interruptions and negative group affects that may squash ideas and innovative approaches to the problem. Once all of the developers have a thorough understanding of the problem it is suggested that such influences will have a reduced affect.

6.3.2 Goal Relationship Identification

One of the benefits of a goal-orientated approach is that it quickly allows identification of relationships that exist between goals [Lamsweerde, 2001]. As we seen from Section 4.5, non-functional goals can be related as being either complementary or conflicting. However, it is also suggested here, that in some cases, two goals – while not being complementary, may also not necessarily be conflicting either. In such circumstances, the two goals represent a different interpretation and emphasis during their implementation, and could be made compatible when they are conceived in a different way and/or their concrete specifics are reworked or re-factored.

As discussed in Section 6.3.1, during the review stages an important consideration for developers will be to recognise the important relationships that may exist between diverse versions. Where the individual developer believes that another developers non-functional goal is complementary or compatible to theirs they should redesign to implement it. However, where the developer believes that another developer's non-functional goal is conflicting with their own they should document their reasons and rationale for why they think it cannot be integrated into their program. After redesign, by each developer, to integrate complementary and compatible non-functional goals of the other developers the design review process begins again - only this time with the redesigned and reintegrated programs. The benefit of such an approach is that it allows a synthesis of non-functional goals to be iteratively integrated and reviewed. More importantly, however, it also begins to unearth particular conflicting aspects of the design problem - where concentration of the contributing developers should be focused. However, such technical conflict shouldn't be viewed negatively, as the discussion of Open-Source suggests in Section 5.3.3, it can stimulate the search and identification of higher-level design solutions.

6.4. Conflict Resolution

It is inevitable, however, that even with iterative stages of review and redesign to find solutions to outstanding non-functional goal conflicts, some goal-conflicts will remain intractable and unresolved. In such situations the prioritization of one goal over another will need to take place. Goal-orientated approaches are highly valuable in such situations

as they facilitate the evaluation of goals with regards to their priority status [Lamsweerde, 2001]. Furthermore, since the emphasis upon certain desirable attributes of dependability is reliant upon the specific user and application domain, it will be necessary to get the input from the user-domain to explicitly decide which dependability attributes are more important than others. Here, also, goal-orientated approaches are useful as they permit an interface for discussion between the technical aspects of the design process and the business/user/management aspects of the user-domain[18] [Lamsweerde, 2001].

The important point to this, is that the user-domain has a direct and explicit input into the prioritisation of what dependability attributes of the CCBS are considered more (or less) important. This makes non-functional prioritisation an explicit user-domain consideration – rather than an implicit (and often unfulfilled) expectation, that is often determined discretely by a single developer during system development. Therefore, such an approach has the additional advantage of making the dimensions and attributes of CBS dependability (and potentially undependability) known to the user-domain prior to deployment.

6.5 Section Summary

The key point of this section is that we can engineer diversity by predisposing developers to pursue different and desirable goals at the non-functional level. This is vitally important since the goals of the process have an overriding influence on the eventual dependability of the CBS. Goal promotion not only helps reduce the complexity of the task faced by developers, but also ensures sufficient diversity and coverage of the important system level attributes that ensure the dependability of CBS. However, because different attributes are important in different user and application domains' it is important to apply an iterative approach that allows progressive problem understanding, synthesis, integration, and evaluation criteria of the subtle interdependencies of non-functional attributes to emerge. Finally, it is vitally important that irreconcilable goal conflicts are explicitly prioritised within the user-domain in which the CBS will operate. This allows

[18] Although outside the scope of this paper, in real-life application, this goal-oriented approach to design-diversity, would probably require an equivalent goal-orientated approach to requirements engineering [cf. Lamsweerde, 2001].

the particular priorities and limits of the CBS dependability dimensions to be made known in advance of deployment.

7. Future Research

The ideas presented by the authors are explorative, in nature, and a number of experimental and empirical possibilities have been discussed. It should first be noted that, due to the inherent range of programming abilities and non-linear scaling of complexity in software engineering, experiments in software design are inherently difficult to control and generalise from. However, the interdisciplinary research project (i.e. DIRC see acknowledgements in Section 9) in which this research will be carried out has resources and facilities that will be useful for conducting such research. Secondly, a number of possible initial research approaches include:

- A pilot experiment into the benefits of increasing human redundancy in fault detection. It is conjectured that a law of diminishing returns will be experienced.

- A pilot experiment into "goal-diversity" using students initially and later attempts to confirm any results through observations within an industrial setting. It is thought that the initial experiment would utilise two groups attempting the same design problem and giving them different (non-functional) goals to pursue (i.e. performance vs. maintainability). Eventual solutions would then be exchanged, discussed, and then merged. Depending upon the outcome of such an experiment, it may be repeated later using three such groups and objectives. With regards to a later industrial investigation, it has been suggested that a study should observe how an organisation and/or traditional software development team manages non-functional goals within the design/development process, in terms of which goals are achieved, maintained, and or compromised.

8. Conclusion

This paper has argued that redundancy and diversity are important ways to achieve dependability in CBS. However, despite the fact that the design process is mostly

responsible for the residual design faults that are to be expected (and may ultimately need to be tolerated) in any non-trivial CBS. There has not been the same exploitation of redundancy and/or diversity principles, at the process level, as that practiced and pursued by fault-tolerant strategies at the computer-system level. Nevertheless, the nature of the Open-Source Software development process begins to suggest that massive forms of human redundancy and diversity can help improve the dependability of CBS through leveraging them for other dependability strategies such as fault prevention, removal and coverage. There are also indications from the Open Source approach that human redundancy and diversity can promote increased problem solving and solution finding via multiple views/ideas and the resolution of self-orientated goal-conflict. In recognition of these Open Source influences, and the important influence of design process goals, the notion of goal-diversity was discussed. It has been suggested that such an approach may improve the dependability of CBS through a process of deliberately predisposing developers to pursue diverse non-functional design goals. It is suggested that this may lead to greater levels of design diversity, coverage, synthesis, solution finding, and integration of desired non-functional attributes that promote the assurance of dependability and trustworthiness of CBS.

9. Acknowledgements

The research of both authors is supported by the EPSRC grant to the Interdisciplinary Research Collaboration on Dependability of Computer-Based Systems; that of ATL is also funded by an EPSRC PhD studentship. Both authors are grateful to many colleagues in “DIRC” for fruitful discussions but would like to single out Denis Besnard and Carles Sala-Oliveras for particularly intensive interchanges on the subjects reported here.

10. References

Anderson, R., (1999) *“How to Cheat at the Lottery (or, Massively Parallel Requirements Engineering),”* in Proc. Computer Security Applications Conference, Phoenix, AZ,

Bell, D. (2000). *Software engineering: A programming approac*h. 3rd edition. Addison-Wesley, U.K.

Beynon-Davies, P., (1999) *Human error and information systems failure: the case of the London ambulance service computer-aided dispatch system project.* Interacting with Computers vol. 11. pp 699–720

Boutin, P., (2002) *Bill Gates Email on Trustworthy Computing*. Online at URL: http://paulboutin.weblogger.com/stories/storyReader$155

Brooks, F. P. (1995). *The mythical man month: Essays on software engineering*. Anniversary Edition. Addison-Wesley, New York, NY.

Byrne, B. (1991) *Social Psychology: Understanding Human Interaction* (6th Edition). Allyn & Bacon Publishers. USA.

Chen, K., (1999). *Identifying the Relationship among Design Methods: Key to Successful Applications and Developments of Design Methods.* Journal of Engineering Design, Vol. 10, No. 2, 1999. pp 125-141.

Cooke, S. & Slack, N. (1991). *Making management decisions*. Prentice-Hall, UK.

Covey, S. R., (1992) *The Seven Habits of Highly Effective People: Powerful Lessons in Personal Change*. Simon & Schuster Ltd. London. UK.

Demarco, T., Lister, T., (1987) *Peopleware: Productive Projects and Teams*. Dorset House Publishing. New York. USA.

Fielding, R. T., (1999) *Shared Leadership in the Apache Project*. Communications of the ACM. Vol. 42, No. 4. April 1999. pp-42-43.

Gacek, C., Lawrie, T., Arief, B. (2001) *The Many Meanings of Open Source*. Department of Computing Science, University of Newcastle-upon-Tyne, Technical Report CS-TR-737

Glass, R.L., (1998) *Software Runaways; Lessons Learned from Massive Software Project Failures*. Prentice Hall New Jersey, USA.

Heylighen, F.H., (2001) *Cybernetics and Second-Order Cybernetics* :in: R.A. Meyers (ed.) *Encyclopedia of Physical Science & Technology* (3rd ed), 2001. Academic Press, New York. USA.

Jackson, M., (2001) *Problem Frames: Analyzing and structuring software development problems.* Addison-Wesley Publishers. Harlow. UK.

Jones, C.B., (2002) *Providing a formal basis for dependability notions.* Department of Computer Science. University of Newcastle. UK. (to be published)

Kandola, R., Fullerton, J., (1998). *Diversity in Action: Managing the Mosaic (2nd Edition).* Institute of Personnel & Development. London. UK.

Knight, J.C., Leveson, N.G. (1986) *An Experimental Evaluation of the Assumption of Independence in Multiversion Programming*. IEEE Transactions on Software Engineering. Vol. 12. No. 1. January., pp 96-109.

Lamsweerde, A. V., (2001) *Goal-Orientated Requirements Engineering: A Guided Tour*. Proceedings of Requirements Engineering 2001, 5th IEEE International Symposium on Requirements Engineering, Toronto, August 2001. pp 249-263.

Lang. R. (2000) *Open Source Software: Issues and Implications*. News @ SEI. Vol. 3. No. 1. Winter 2000., pp. 6-7. Online at URL: http://www.sei.edu

Laprie, J. C., (1995) *"Dependable Computing: Concepts, Limits, Challenges,"* in *25th* IEEE International Symposium on Fault-Tolerant Computing - Special Issue, pp. 42-54, Pasadena, California, USA, IEEE.

Laprie, J.C., (1992) (Ed.). *Dependability: Basic concepts and terminology — in English, French, German, Italian and Japanese,* Dependable Computing and Fault Tolerance. Vienna, Austria, Springer-Verlag

Latham, G.P., Locke, E.A., (1979) *Goal-Setting – A Motivational Technique That Works.* Organizational Dynamics, Vol. 8. No 2. pp 68-80.

Lewontin, R., (1982). *Human Diversity*. Scientific American Books. San Francisco. USA.

Mason, D., Willcocks, L., (1994) *Systems Analysis, Systems Design.* Alfred Waller Publishing, Oxfordshire, UK.

McPhee, K., (1997) *Design Theory and Software Design*, Technical Report TR 96-26 (October 1996 : - Revised May, 1997) Department of Computer Science, University of Alberta, Edmonton, Alberta Canada.

Meyer, B., (2000) *The Ethics of Free Software*. Software Development Magazine. Online at URL: http://www.sdmagazine.com/articles/2000/0003/0003d/0003d.htm

Moody, G., (2001) *Rebel Code: Linux and The Open Source Revolution*. Allen Lane Penguin Press. Hammondsworth Middlesex. UK.

Pettit, K., Chen, S., Coffing, C., Ho, T., Brockmeier, J., Harris, A., (2000) *Suse Linux: Install, Configure, and Customize*. Prima Publishing, California. USA.

Popov, P., Strigini, L., Romanovsky, A., (1999) *Choosing Effective Methods for Design Diversity – how to progress from intuition to science*. Lecture Notes in Computer Science, Vol. 1698.

Prata, S., (1995) *C++ Primer Plus* (2nd Edition). Waite Group Press. California. USA

Randell, B. (2000). *Turing Memorial Lecture: Facing up to faults*. The Computer Journal, Vol. 43. No. 2. pp 95-106.

Raymond, E.S. (1999). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Rielly & Associates, Inc. USA

Sanders, J., (1998) *Linux, Open Source, and Software's Future*. IEEE Software. September/October 1998. pp 88-91.

Shepard, C, R., (1964). *Small Groups*. San Francisco, CA. Chandler Publishing. USA :in: Kandola, R., Fullerton, J., (1998). *Diversity in Action: Managing the Mosaic (2nd Edition).* Institute of Personnel & Development. London. UK. Cited pp 49.

Sommerville, I., (2001) *Software Engineering* (6th Edition). Addison-Wesley Publishers. Essex. UK.

Viller, S., Bowers, J., Rodden, T., (1997) *Human Factors in Requirements Engineering: A Survey of Human Sciences Literature Relevant to the Improvement of Dependable Systems Development Processes.* Technical Report CSEG/8/1997. Computing Department. Lancaster University. UK.

Weinberg, G. M. (1971). *The psychology of computer programmin*g. Van Nostrand Reinhold, London.

Weinberg, G.M., Schulman, E.L., (1974) *Goals and Performance in Computer Programming.* Human Factors. Vol. 16. No. 1.pp 70-77.

Westerman, S. J., Shryane, N. M., Crawshaw, C. M. & Hockey, G. R. J. (1997). *Engineering cognitive diversity*. in F. Redmill & T. Anderson (Eds). *Safer System*s. Proceedings of the 5th Safety-critical Systems Symposium, Brighton, UK (pp. 111-120).

Yamouchi, Y., Yokozawa, M., Shinohara, T., Ishida. T. (2000). *Collaboration with Lean Media: How Open-Source Software Succeeds*. Conference Paper Presented at Computer Supported Cooperative Work Conference 2nd to 6th December 2000. Philadelphia., pp 329-338.