Toward a Science of Design for Software-Intensive Systems

***Joshua Eckroth*[1]*, Ricardo Aytche*[2] *and Guy-Alain Amoussou*[3]**

This research is intended as an initial step toward the development of a rigorous science of design for software-intensive systems. Our work based on existing research, has provided new definitions for *design*, *science of design*, and *software-intensive systems*. By identifying design issues currently affecting the field, we will show why software-intensive systems require a science of design. We have also provided some functional requirements for a science of design: what it must address and how it might do so. We anticipate that these definitions and ideas will be embraced in the research community, and utilized as a foundation for continued investigations. This work is a starting point for future research in this area with extensive references and ideas for further research included.

keywords: science of design, software-intensive systems, software engineering, design theory, software defects

1 Introduction

Software is becoming more pervasive, a result of the proliferation of software-intensive systems, or systems whose main functionalities are implemented via software (see our more specific definition in section 2). As such, software engineering is increasingly important, not only because software must be engineered, but because software is typically and historically engineered poorly (issues with software engineering are discussed in section 3). We will incorporate a multitude of resources to aid in creating an encompassing definition for design, science of design, and software-intensive systems, thus identifying the most common design problems relevant in software development, and suggesting possible solutions. In doing so, the process of making the science of design a teachable science will begin, whose fundamental ideas can be extracted for the improvement of design processes in all disciplines. If we can improve software engineering techniques, we can improve the software on which we depend.

Essential to any engineering discipline is design. Herbert Simon argued, “Engineers are not the only professional designers. Everyone designs who devises courses of action aimed at changing existing situations into preferred ones” (Simon 1996 p. 111). Clearly software engineering contains all the elements of design. Additionally, all steps in the software engineering lifecycle contain elements of design (see the software engineering lifecycle in section 3). Clearly, Simon considered the notion of a *science of design* which studies the techniques and processes of design itself, as an aim to improve them (see the expanded definition in section 2). If science of design can inform how design is enacted, and show how to improve the techniques and processes of design, then software engineering as a whole will improve as will our software. The purpose of this report is to consider what such a *science of*

1 Humboldt State University. email: `jre9@humboldt.edu`
2 University of Central Florida. email: `ri945867@pegasus.cc.ucf.edu`
3 Humboldt State University. email: `amoussou@humboldt.edu`

design for *software-intensive systems* should include:
what we are trying to improve by way of definitions (section 2); what issues exist in the engineering of software-intensive systems (section 3); why a science of design will improve said engineering issues and what such a science might contain as its queries, body of knowledge, and results (section 4).

2 Definitions

Important to any field of inquiry are standardized definitions of certain terms. In our case, we find it necessary to propose definitions for *design, science of design,* and *software-intensive systems*, as previously there have been no consistent definitions of these terms.

Design, as a verb, has been an elusive word, as noted by many authors, and in particular Love (2002): "[design has] different meanings in different domains, [it is] used in different ways by researchers in the same domain, and [is] found in the literature referring to concepts at different levels of abstraction." Willem (1990) has also pointed out that "the activity of designing has been defined largely by association." That is, design is often defined in terms of the context or previous knowledge a researcher possesses. We are interested in the design of software-intensive systems; however we wish to define design in a broader sense. The traditional definition is provided by Simon, as noted above. Perhaps such a definition, beginning with "everyone who devises a course of action," is too broad. Friedman (2003) has identified three attributes most definitions of design share: design is a process; the process is goal-oriented; and the goals are meeting needs, improving situations, or creating something new. Langlois (2003) defines design as "the product of a process of *designing* of that desired artifact, system, process." Mirroring Simon's distinction between the natural sciences and the artificial sciences (pp. 1-5), Fischer (2003) defines design as the activity that pursues the question of "how things ought to be" rather than "how things are." Taking into account the above viewpoints, we define design as the following:

> Design (as a verb) is a human activity resulting in a unique design (specification, description) of artifacts. Therefore, what can be designed varies greatly. However, common to all design is intention: all designs have a goal, and the goal is typically meeting needs, improving situations, or creating something new. Thus, design is the process of changing an existing environment into a desired environment by way of specifying the properties of artifacts that will constitute the desired environment; in other words, creating, modifying, or specifying how to create or alter artifacts to meet needs. In addition, it is best communicated in terms of a particular context, as previous knowledge, experience, and expectations play a strong role in designing and understanding designs.

It is important to add that design typically involves creativity, coping with complexity, and group dynamics because design is often done in teams, knowledge-sharing, and ethics, as explored in section 4.

We utilize Cross's definition of the *science of design,* which itself is a unification of others' definitions, including Simon's original conception. Simon described it as "a body of intellectually tough, analytic, partly formalized, partly empirical, teachable doctrine about the design process" (1996 p. 113), while Cross says the science of design "refers to that body of work which attempts to improve our understanding of design through 'scientific' (i.e., systematic, reliable) methods of investigation" (2001).

Though profound in context, it should be noted that the above definition of the science of design is not all inclusive. It contains many "loaded" words: intellectually tough, empirical, teachable, and scientific. In order to increase the strength of their definition, formalization of some of these terms is desirable.

“Intellectually tough” is perhaps a side-effect of an important discipline, but nevertheless the intention is to qualify the science of design as rigorous and analytic, whose statements must be falsifiable. In this vein, if the science of design is to utilize scientific methods of investigation, it must be rigorous and analytical. To gain more insight into the nature of scientific methods, we refer to a definition of *science of design,* culled from the dictionary definitions of *science* and *design*:

> The observation, identification, description, experimental investigation, or theoretical explanation covering general truths or the operation of general laws necessary to systematically formulate and execute a creative plan, while satisfying functional requirements on performance and resource usage, to produce quality results, in an artistic or highly skilled manner (Heritage 2000).

The methods of investigation we will be using in a science of design must inherently include observation, identification, investigation, and so on. We expect our improved understanding of design, and scientific methods will enable us to systematically formulate and execute design techniques that will satisfy design requirements in an artistic or skilled manner. For reasons detailed in section 4, our scientific techniques will model more social sciences rather than “hard” or technical sciences. In all, we aspire such a science to be teachable, which requires a mostly-agreed upon compilation of beliefs and knowledge about design; teachableness will be addressed as a further research direction in section 6.

For our purposes, we define a *software-intensive system* to be a computational artifact whose functionality is provided mostly by software rather than hardware (Kossiakoff and Sweet 2002; Putnam 2000). “Such systems may very well include elements of hardware and people, and the connections to them and how to include consideration of them in the overall design of the system is, of course, critical” (Freeman 2004). The following are some examples: global-positioning devices consist of a collaboration of satellites, electromagnetic transmitters and receivers, processors, timing and much more, which is all controlled by software embedded systems. The software utilized in GPS systems must operate with precision, collectively and accurately to produce a required result, with minimal input from the user to be considered effective. Cell phones are much more than just modes of communication. They incorporate online access, memory capacity, text messaging, photo and video capabilities in addition to placing phone calls. The embedded systems in cell phones are also software-intensive because all functions have to communicate and inter-operate with each other with minimal or no failures to be considered effective by users. Other examples are modern mobile media players, and any software product.

The design issues of software-intensive systems are mostly the same as software systems; in other words, software engineering is a prevalent factor in engineering software-intensive systems, therefore the techniques of that discipline apply. However, since software-intensive systems must include a significant amount of hardware to successfully operate the software, extra engineering concerns exist. Such concerns include how the software interacts with the hardware (such as control and timing issues), how the software is stored within the hardware, and the degree of portability of the software. Hence, engineering software-intensive systems requires a framework that is more expansive than that offered by the current software engineering field. But, again as Peter Freeman stated in his article eliciting interest in a science of design for software-intensive systems, “the core consideration must be the software aspect” (2004).

Now that we have presented definitions of design, science of design, and software-intensive systems, which we believe will be agreed upon by the research community, we will discuss why a science of design of software-intensive systems is necessary.

3 Design Issues of Software-Intensive Systems

To effectively argue that designing software-intensive systems will benefit from a scientific analysis of such a process, we will convincingly demonstrate that there exist problems with the current techniques of design. Problems exist because results are too often insufficient. Additionally, we intuitively believe that improving the design of systems will improve their quality.

The software lifecycle provides a convenient order of explanation of the issues in designing software-intensive systems:

- Requirements elicitation
- Solution generation (often described as "design")
- Implementation
- Testing
- Deployment
- Maintenance

At this point, we anticipate concern over the second step of the lifecycle which is typically called "design." Why is the entire lifecycle considered if only the second step involves design? Our definition of *design* refers to creating artifacts to meet needs. When creating software-intensive systems, the system is the "artefact" that meets the need, and determining the nature, specification or "design" of this artifact is typically achieved by the process described as the "software lifecycle." Each step indeed involves design, but taken as a whole, we have a design process that describes the creation of artifacts. Freeman (2004) mentions "we are really concerned with all the stages of activity involved in conceptualizing, sketching, framing, implementing, commissioning, and ultimately modifying a complex system involving software – not just the stage following specification and preceding programming [...]"

The first step is requirements elicitation. Requirement specifications are generated by stakeholders (individuals who have a vested interest in the existence and quality of the final product), and communicated to software engineers for the production of a software package. Once guidelines have been established, engineers then work to meet these criteria when architecting and coding software. This appears to be a simple feat, but experienced software developers recognize that it is impossible to produce a bug-free program. The constant pressure of time and budget limitations often force programmers to produce low-quality software. This occurs when pressure received by project managers is disseminated to the programmers, which independent studies prove causes minor bugs to be overlooked. "Over half of software bugs are not found until downstream in the development process, leading to significant costs…a bug which costs \$1 to fix on the programmer's desktop costs \$100 to fix once it's incorporated into a complete program, and many thousands of dollars if it is identified after the software has been deployed in the field" (NIST 2002).

A theory classified as "requirement tracing" was documented in the proceedings of the Joint 10th European Software Engineering Conference (ESEC) in Lisbon, Portugal, to help minimize the repercussions of the above problem. "Implementing *requirements tracing*, where developers keep track of how requirements are being implemented in source code, and how changes in requirements change the source code, helps developers and project managers discover potential conflicts between code components, better understand the effect of changes in requirements, and meet quality-control specifications" (Heindl and Biffl 2005). Without employing such a process, it would be quite easy to produce a software failure. Software failures can be classified into two broad categories, the first dealing with the inability of the implemented software to perform to the expectations of the users, and the second dealing with the inability of the software developers to produce a working or functioning system for the user (Ewusi-Mensah 2003). This will be further investigated later in this section.

As coding begins, programmers are put to the task of translating mental solutions into coded solutions. This "concept assignment problem" identifies how high-level concepts are written in and associated with code. This problem is encountered during initial coding and during maintenance as well, when another's code must be comprehended before bugs can be fixed. When a programmer revisits code, this problem must be faced; the programmer must identify how the code relates to the functionality addressed (Robillard 2005).

During implementation, the software should be coded in components that enable pieces of code to be reused throughout the software and in other projects. Reusable component design "is not a goal in itself; it aims at speeding up and decreasing maintenance costs" (Estublier and Vega 2005). Developing reusable components is more costly than not and as such, reusable components must be successful to be worth developing. Success is typically measured by two factors: scope of reuse and reuse ratio in target application (Estublier and Vega 2005). Added difficulty results from the fact that most software is legacy software, and was not developed with a *component-based development* framework, which maximizes software reusability. Le Gear and Buckley propose a method that helps "encapsulate the information abstraction you wish to reuse" in legacy software, which further allows creation of a wrapper which can then be interfaced into new software (2005).

When software-intensive systems are produced, one major factor for the final product is dependability. "It is increasingly argued that uncertainty is an inescapable feature of the design and operational behaviour of software-intensive systems" (Littlewood et al. 1995). Though many resources validate this statement, there are certain precautions that can reduce system malfunctions.

Ensuring dependability requires sufficient testing. It appears that rigorous testing and re-evaluation of software is excessively ambiguous and tedious. A suggested solution for this problem would be to devise a computational method to replace current empirical methods. This evolution would track improvements in understanding the fundamental properties of engineering subject matter. For example, the Wright brothers experimented with wind tunnels to evaluate airfoils, a task carried out today by computational fluid dynamics. Computational methods can give confidence when stakes are high and engineering questions must be objectively decided. This theorized evolution of software engineering towards computational semantics could facilitate the production of intelligent tools to help determine trustworthiness in a definitive way guaranteeing a standard for trustworthy software. An effort is underway to develop a computation tool for analysing the functionality of malicious code as a small step towards answering this question (Linger 2004).

Regarding overall scheduling and planning, we see that organizations in both the public and private sectors have too often been forced to cancel software projects that have far exceeded the original cost and schedule or time-of-completion estimates, or failed to achieve the desired minimal functionalities. The cost of such project cancellations have been staggering in recent years, owing in part to the increased size and complexity of new software technology (Ewusi-Mensah 2003). Stakeholders unrealistically expect a precise estimate of time and cost of software development projects. But apart from the fact that software estimates are typically wrong, the term "estimate" is ambiguous: managers often expect estimates to be met regardless of how development proceeds, while developers often give best-case estimates (Aranda and Easterbrook 2005).

Another key issue for analysing cost-effective production is the measurement of effort from all developers on the project. But measuring programmer effort becomes difficult when projects become large or programmers are working in "crunch-time." Hochstein et al. combine self-reporting and automatic techniques to better evaluate programmer effort (2005).

Finally, when an organization decides to cancel, or "abandon," a project later in the software development process, it can be reasonably surmised that the project was a failure because the organization did not achieve its original objective with respect to the project. Types of abandonment are: total abandonment, the most frequent case, where there is complete termination of all activity on the software project prior to full implementation; substantial abandonment, when there is a major truncation or simplification of the project prior to full implementation, making it radically different from the original specifications; and partial abandonment, a reduction of the original scope of the project, without entailing major changes to the project's original specifications prior to full implementation (Ewusi-Mensah 2003).

A broader more ambiguous concern of developing software-intensive systems is ensuring trust. Software system developers too often focus on schedule and cost, resulting in performance and functional technical requirements to become consuming issues. Rarely is trustworthiness considered. Though often overlooked, trust is an issue that needs to be addressed, for it helps software designers not only to consider how the software will perform, but also to account for the consequences of failures (Bernstein 2004). Trustworthy software is stable software. It is sufficiently fault-tolerant, it does not crash at minor flaws, and will shut down in an orderly way in the face of trauma. Trustworthy software does what it is supposed to do, and can repeat that action, always producing the same kind of output from the same kind of input (Bernstein 2004). In an ideal world, trustworthy software would carry absolute guarantees that the software will perform its required functions under all possible circumstances, will do so on time, and will never perform any action that has hazardous consequences. Unfortunately, such absolute guarantees are impossible (Berzins 2004).

We believe these issues will be best addressed by a science of design for software-intensive systems.

4 Science of Design for Software-Intensive Systems

The famed NATO conference of 1968 about software engineering was one of the first discussions of the need of a scientific approach to developing software (Naur and Randell 1969). Since that time there has been relatively little research into developing a scientific approach, but there has been a recent revival as demonstrated by the NSF Program Solicitation NSF 05-620. The motivation for considering a science of design for software-intensive systems is well documented by this NSF Program funding this research.

> "Research on software systems typically focuses on how to reduce errors, increase reliability under duress, make systems easier to learn and use, isolate failures and reduce their impact, adapt to new purposes, contexts, and operational environments, and so on. The Science of Design Program seeks ideas that will broaden the ways in which this research is conducted, particularly in light of increasing software sophistication, diversity, dependencies, and risk. The focus of design as the central theme of this program is intended to raise the level of discourse, generate new perspectives, and take a more holistic view of the major challenges in constructing software-intensive systems." (NSF Program Solicitation NSF 05-620)

Further, "the objective of the program is to bring new paradigms, concepts, approaches, models, and theories into the development of a strong intellectual foundation for software design, which will ultimately improve the processes of constructing and modifying software-intensive systems." Mirroring our survey of definitions of *science of design*, the Program Solicitation states "this body of knowledge needs to be intellectually rigorous, formalized where appropriate, supported by empirical evidence when possible, and above all, teachable." As stated previously, these are our intentions.

Software engineering as a profession is still trying to define itself (Grimson and Kugler 2000; Griss 1998; Yang and Mei 2006). Lewerentz and Rust's "Are software engineers true engineers?" (2000) recognizes that common views of engineering contain elements of "application of theories from the sciences for the construction of artifacts," "systematic development processes," "quantitative measurements and quality control," etc. The authors determined software engineering does contain these stereotypical elements of other engineering professions, but also many properties of the social sciences in greater importance than other engineering fields.

This is partly the nature of our research, as discussed in this section. A *science of design* for software-intensive systems is a wider abstraction than a study of *software engineering*. However, the two terms overlap considerably; consider Sommerville's (1996) definition of software engineering: "concerned with theories, methods and tools which are needed to develop software for ... computers," (quoted in Bryant 2000b), compared with our definition of science of design: "the observation [...] or explanation covering [...] laws necessary to [...] execute a creative plan, while satisfying functional requirements [....]" The notions are similar, and the debate of what constitutes *software engineering* continues. Our research in particular addresses which processes of software engineering need improvement and how to begin inquiries for new directions. Also of note, much literature has distinguished "natural sciences" from "artificial sciences," defining the former as descriptive and the latter as normative. We are clearly in the normative viewpoint, as software is artificial, but our intention of using "scientific methods" brings us partly back to descriptive analyses. This duality of a science of design is explored by Kroes (2002). In this paper, we will "straddle the fence", so to speak, applying descriptive analysis and normative analysis when applicable.

Design involves many facets; first, consider "presence of creativity." Science of design must address the role of creativity, since most "good" designs are deemed creative. Dorst and Cross (2001) noted that judgments of creativity are often consistent, and that more time spent on the problem formulation yields more creative results. Interestingly, they believed that creative design involves formulating the problem, thinking of a solution attempt, then reformulating the problem to better match the solution attempt, which would then itself demand a new solution attempt. Hence, the problem and solution co-evolve until a perfect match is made. Li et al. (2006) explained some characteristics of creative designers: they are chance-takers, they are motivated by the desire to be creative, and they do not rely solely on knowledge or past experience. Bonnardel (1999) explores the importance of "analogy-making" in creative design, Fischer (2003) relates the need for collaboration, and Standler (1998) suggests finding alternative ways to view phenomena or ask a question.

Most design is a team effort, and software designs in particular follow peculiar laws of team dynamics. In *The Mythical Man-Month* (Brooks 1975), expounds on such laws, and a science of design for software-intensive systems must consider them.

Another important design component is the issue of complexity: modern software systems are exceedingly complex and designers must be able to cope with "the more difficult the problem, the more difficult the solution" (Glass 2002). In addition, Glass has found that "for every 25% increase in problem complexity, there is a 100% increase in complexity of the software solution," and "explicit requirements explode by a factor of 50 or more into implicit (design) requirements as a software solution proceeds." Complexity of design is not just a measure of the complexity of the problem. In "A model of factors influencing the design requirement," Darlington and Culley (2004) note that not only does the nature of the technical product add complexity (mechanical systems are less complex than electronic systems, that they showed), but "no matter what the actual complexity of the product the circumstances surrounding a design episode will have a bearing on what might be termed *effective* complexity." Two example circumstances were provided: "the character of the solution domain" and "the experience that a particular company has in respect of a new product" will affect *effective complexity*. Further investigation should consider the processes of componentization and unification, where a difficult problem is broken into easier sub-problems, in a divide-and-conquer technique. This technique can be applied both in software design and team makeup. It may also be useful to study how limiting innovation, as in engineering design, versus limiting constraints, as in architecture design, might reduce the complexity of the problem domain.

Design representation and validation are essential to science of design, in the former for inter-disciplinary collaboration, which has been noted by (Darlington and Culley 2004) to contribute to creativity, but also in the latter to convince others that a design representation matches reality. Although Simon believed abstract representations of design were possible (1996 pp. 131-134), Baldwin (2003) argues that such existing, abstract methods of representation do not enable comparison of designs across inter-disciplinary domains. Baldwin introduces requirements for a useful representation schema: the representation must actually reflect the design in its domain, but it must also verifiably present the design. That is, a representation schema must have a means of validating its representations; Vahidov (2006) mirrors these sentiments. To this end, software engineering has provided methods like UML for representing software logic and story-boards for representing an interface. Further review may yield information that tests the inter-disciplinary effectiveness of these and other representation schemas.

In order to verify that designs are useful real solutions, and that the techniques employed are effective, an evaluation framework must be available to assure that the design methods used actually produce good designs. Patterns of good techniques may eventually develop; Shaw (1994) provides a starting point for such patterns.

What is often considered the "design process" is probably the most important component of a science of design. Apart from the software lifecycle mentioned above, there are several opinions of what the typical or ideal engineering design process contains. For example, Vidosic (1969) defined the following stages:

- Recognition
- Definition
- Preparation
- Conceptualization
- Synthesis
- Evaluation
- Optimization
- Presentation

Dixon divided the engineering design process into three components: inventiveness, engineering analysis, and decision making (1966). Takeda et al. (1990) described a design process that evolves step-wise, with each step containing the elements *awareness of problem*, *suggestion*, *development*, *evaluation*, and *conclusion* (Yoshioka et al. 1993). These steps repeat as necessary, solving small attainable goals or problems of a greater goal.

The methods of software engineering can be seen as formulations of a design process, with the software lifecycle above mostly representing the waterfall method. But other methods are becoming popular, such as extreme programming and agile development. These new approaches change the design process, making it more reflexive and iterative incorporating the more cyclic processes of Takeda et al. (1990).

We propose the following design process. First, the problem must be formulated, and the designer should ask "what makes this a problem?" and "in what context does this problem exist?" Then, solution requirements must be determined, distinguishable between qualitative and quantitative requirements. The former are value-judgments, like "the software must perform efficiently and responsively," while quantitative requirements are measurable, such as "the software package must fit on one CD-ROM, and require no more than 64 MB of memory." Then, to cope with complexity, the problem may be reduced or divided, and each "sub-problem" iterated upon the beginning of this process. Next, the designer finds obstacles to using the most obvious solution, if any exist. Recall that Dorst and Cross (2001) showed that the problem statement and solution attempts co-evolve, therefore at this point, when the designer has considered a solution attempt but discovered it is insufficient, the problem may be reformulated. The method of judging the success of solution attempts requires an evaluation framework, as described above. In order to generate solution attempts, one might consider past solutions, perhaps utilizing analogies from other domains. After sub-problems are resolved, their solutions must be unified to solve the larger problem, where upon this unified solution is evaluated. If all goes well, and the complete problem has been solved, it is packaged and deployed. Maintenance issues now come into play, and the software-intensive system enters the phase of product evolution, or dynamic update.

The scientific methods science of design might utilize will not be those of the "hard" sciences, but rather surveys, observation of individuals and groups, qualitative research, content analysis, and other techniques used by the social sciences (Chadwick et al. 1984). Design is a psycho-social activity, rather than a natural phenomenon, so research methods of psychology and sociology are more relevant.

This overview of the components of science of design for software-intensive systems is not exhaustive, but meant as a starting point or launch for future research.

5 Conclusion

This research is intended as an initial step toward the development of a rigorous science of design for software-intensive systems. Our work has provided new definitions of *design, science of design,* and *software-intensive systems* that are not original but rather the unification of many other ideas. In addition, we have stated why software-intensive systems need a science of design, by identifying design issues currently affecting the field. We have also provided some requirements for a science of design: what it must address and how it might do so. In future work, it is our hope and belief that these three accomplishments will be utilized and a science of design for software-intensive systems will start to take shape; a morphing of its own body of knowledge and experience, practitioners, and curriculum.

6 Further Research

First and foremost, several resources on software engineering in general have been located but not included in our research. They include (Boehm 2006) and (Bryant 2000a). Utilizing middleware has become an important technique, and (Emmerich 2000) provides a roadmap for future directions.

Design methodology research is a strong component of the science of design, for any field. Useful directions are (Cottam et al. 2002); (Cross 1984); (Houkes et al. 2002); (Kroes 2002); (Lawson 1997).

Research on creativity in design may benefit from (Gero 1996), (Nguyen and Swatman 2005), and (Vidal, et al. 2004). Inter-disciplinary factors in design representation schemas are addressed in (Hendry 2004), and the conceptual process of design in inter-disciplinary teams is highlighted in (Austin et al. 2001). Also regarding representations of designs, Chakrabarti and Bligh (2001) provide insights into functional reasoning methods embedded in representation schemas. Design requirement capture relates to representation schemas, as the design requirement is a representation of a design and must be evaluated and understood in the same manner as expressed above. (Darlington and Culley 2004) provides a good reference of factors influencing the capture of design requirements.

There exists a distinction in software design between software engineering versus software architecture, mirroring that of civil engineering versus civil architecture. Software architecture designs systems, in terms of their organization, while software engineering implements designs. The following resources may provide a starting point for incorporating the field of software architecture into the present discussion: (Albin 2003); (Bass et al. 2003); (McBride 2004); (Perry and Wolf 1992); (Sewel and Sewell 2002); (Shaw and Garlan 1996).

The science of engineering design, rather than software design, can be found in these sources: (Barratt 1980); (Bernsen 1983); (Bucciarelli 1994); (Bucciarelli 2002); (Ertas and Jones 1996); (Glegg 1973); (Hill 1970); (Maier 1977); (Petroski 1994).

Software-intensive systems are inherently systems, and an understanding of general systems theory will benefit from their study. These references provide such a background: (Gall 1977); (Skyttner 2001); (Weinberg 1975); (Weinberg 1982).

The role of ethics though not explored here is very important, particularly since software engineering is becoming a profession (Bott 2001). “Ethics in computer software design and development” (Thomson and Schmoldt 2001), as well as Bott (2001), demonstrate the need and relatively little attention paid to ethics in software development.

Finally, a very important consideration that has not been developed in our research is the issue of effective teaching. If we are speaking of a science, it must be teachable. So a science of design for software-intensive systems will have a body of knowledge and techniques that can be elucidated in the classroom creating new generations of designers. Such curriculum will become reality as science of design matures.

7 References

(Albin 2003) Albin, Stephen T. *The art of software architecture: design methods and techniques.* Wiley Publishers: Indianapolis, IN. 2003.

(Aranda and Easterbrook 2005) Aranda, Jorge and Steve Easterbrook. “Anchoring and Adjustment in

Software Estimation." *Proceedings of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13).* Sepetember 5-9, 2005. Lisbon, Portugal.

(Austin et al. 2001) Austin, Simon, John Steele, Sebastian Macmillan, Paul Kirby and Robin Spence. "Mapping the conceptual design activity of interdisciplinary teams." *Design Studies* 22(3): 211-232. 2001.

(Baker, et al 2005) Baker, Paul, Paul Bristow, Clive Jervis, David King, Robert Thomson, Bill Mitchell, Simon Burton. "Detecting and Resolving Semantic Pathologies in UML Sequence Diagrams." *Proceedings of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13).* Sepetember 5-9, 2005. Lisbon, Portugal.

(Barratt 1980) Barratt, Krome. *Logic and design: the syntax of art, science & mathematics.* Eastview Editions: Westfield, NJ. 1980.

(Bass et al. 2003) Bass, Len, Paul Clements, and Rick Kazman. *Software architecture in practice.* Addison-Wesley: Boston. 2003.

(Bernsen 1983) Bernsen, Jens. *Design, the problem comes first.* Danish Design Council: Copenhagen. 1983.

(Bernstein 2004) Bernstein, Larry. "Trustworthy Software Systems." *Proceedings of the Center for National Software Studies Workshop on Trustworthy Software* (NPS-CS-04-006). 2004.

(Berzin 2004) Berzin, Valdis. "Trustworthiness as Risk Abatement." *Proceedings of the Center for National Software Studies Workshop on Trustworthy Software* (NPS-CS-04-006). 2004.

(Boehm 2006) Boehm, Barry. "A View of 20th and 21st Century Software Engineering." *ICSE '06.* Shanghai, China, May 20-28, 2006.

(Bonnardel 1999) Bonnardel, Nathalie. "Creativity in design activities: The role of analogies in a constrained cognitive environment." *Proceedings of the 3rd Conference on Creativity & Cognition.* Loughborough, UK. 1999.

(Bott 2001) Bott, Frank. *Professional issues in software engineering.* Taylor & Francis: London. 2001.

(Brooks 1975) Brooks, Jr., Frederick P. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley Publishing Company, Inc. 1975.

(Bryant 2000a) Bryant, Antony. "'It's Engineering Jim ... but not as we know it.': Software Engineering – solution to the software crisis, or part of the problem?" *ICSE 2000.* Limerick, Ireland, 2000.

(Bryant 2000b) Bryant, Antony. "Metaphor, myth and mimicry: The bases of software engineering." *Annals of Software Engineering* 10: 273-292. 2000.

(Bucciarelli 1994) Bucciarelli, Louis L. *Designing engineers.* MIT Press. 1994.

(Bucciarelli 2002) Bucciarelli, Louis L. "Between thought and object in engineering design." *Design Studies* 23(3): 219-231. 2002.

(Chadwick et al. 1984) Chadwick, Bruce A., Howard M. Bahr, Stan L. Albrecht. *Social Science Research Methods.* Prentice-Hall: Englewood Cliffs, NJ. 1984.

(Chakrabarti and Bligh 2001) Chakrabarti, Amaresh and Thomas P. Bligh. "A scheme for functional reasoning in conceptual design." *Design Studies* 22(6): 493-517. 2001.

(Cottam et al. 2002) Cottam, M., M. G. Hodskinson and I. Sherrington. "Development of a design strategy for an established semi-technical product. A case study of a safety harness for tree workers." *Design Studies* 23(1): 41-65. 2002.

(Cross 1984) Cross, Nigel (Ed.). *Developments in design methodology.* Wiley: Chichester. 1984.

(Cross 2001) Cross, Nigel. "Designerly Ways of Knowing: Design Discipline Versus Design Science." *Design Issues* 17(3): 49-55. 2001.

(Darlington and Culley 2004) Darlington, M. J. and S. J. Culley. "A model of factors influencing the design requirement." *Design Studies* 25(4): 329-350. 2004.

(Dixon 1966) Dixon, John R. *Design Engineering: Inventiveness, Analysis, and Decision Making.* McGraw-Hill Book Company. 1966.

(Dorst and Cross 2001) Dorst, Kees and Nigel Cross. "Creativity in the design process: co-evolution of problem-solution." *Design Studies* 22(5): 425-437. 2001. (Emmerich 2000) Emmerich, Wolfgang. "Software Engineering and Middleware: A Roadmap." *Future of Software Engineering*. Limerick, Ireland. 2000.

(Ertas and Jones 1996) Ertas, Atila and Jesse C. Jones. *The engineering design process.* John Wiley & Sons: New York. 1996.

(Estublier and Vega 2005) Estublier, Jacky and German Vega. "Reuse and Variability in Large Software Applications." *Proceedings of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13).* Sepetember 5-9, 2005. Lisbon, Portugal.

(Ewusi-Mensah 2003) Kweku Ewusi-Mensah. *Software Development Failures*. MIT Press. 2003.

(Fischer 2003) Gerhard Fischer. *Coping with Software-Intensive Systems—Design, Design Communities, Meta-Design and Social Creativity.* Position Paper for the NSF Invitational Workshop "Science of Design: Software-Intensive Systems". 2003.

(Freeman 2004) Freeman, Peter A. "Science of Design." *Computing Research News* 16(1): 4-7. 2004.

(Friedman 2003) Friedman, Ken. "Theory construction in design research: criteria: approaches, and methods." *Design Studies* 24(6): 507-522. 2003.

(Gall 1977) Gall, John. *Systematics: how systems work and especially how they fail.* Quadrangle / New York Times Book Co.: New York. 1977.

(Glass 2002) Glass, Robert L. "Sorting Out Software Complexity." *Communications of the ACM* 45(11): 19-21. 2002.

(Glegg 1973) Glegg, Gordon Lindsay. *The science of design.* Cambridge Engineering University Press.

1973.

(Gero 1996) Gero, John S. "Creativity, Emergence, and Evolution in Design." *Knowledge-Based Systems* 9(7): 435-448. 1996.

(Grimson and Kugler, 2000). Grimson, J.B. and H. Kugler. "Software needs engineering: a position paper." In *Proceedings of the 22nd International Conference on Software Engineering*. June 4-11, 2000. Limerick, Ireland.

(Griss 1998) Griss, M. "Software engineering as a profession: industry and academia working together." In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. November 1-5, 1998. Lake Buena Vista, Florida.

(Heindl and Biffl 2005) Heindl, Matthias and Stefan Biffl. "A Case Study on Value-based Requirements Tracing." *Proceedings of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13).* Sepetember 5-9, 2005. Lisbon, Portugal.

(Hendry 2004) Hendry, David G. "Communication Functions and the Adaptation of Design Representations in Interdisciplinary Teams." *DIS2004*, Cambridge, MA. August 1-4, 2004.

(Heritage 2000) "The American Heritage Dictionary of the English Language". Houghton Mifflin Company. 2000.

(Hill 1970) Hill, Percy H. *The science of engineering design.* Holt, Rinehart and Winston: New York. 1970.

(Hochstein, et al 2005) Hochstein, Lorin, Victor R. Basili, Marvin V. Zelkowitz, Jeffrey K. Hollingsworth, Jeff Carver. "Combining Self-reported and Automatic Data to Improve Programming Effort Measurement." *Proceedings of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13).* Sepetember 5-9, 2005. Lisbon, Portugal.

(Houkes et al. 2002) Houkes, Wybo, Pieter E. Vermaas, Kees Dorst, and Marc J. de Vries. "Design and use as plans: an action-theoretical account." *Design Studies* 23(3): 303-320. 2002.

(Kossiakoff and Sweet 2002) Kossiakoff, Alexander and William N. Sweet. *Systems Engineering Principles and Practice.* Wiley-IEEE, 2002. p. 91.

(Kroes 2002) Kroes, Peter. "Design methodology and the nature of technical artefacts." *Design Studies* 23(3): 287-302. 2002.

(Langlois 2003) Richard N. Langlois. *Science of Design: Software-Intensive Systems.* Position Paper for the NSF Invitational Workshop "Science of Design: Software-Intensive Systems." 2003.

(Lawson 1997) Lawson, Bryan. *How designers think: the design process demystified.* Architectural Press: Oxford. 1997.

(Le Gear and Buckley 2005) Le Gear, Andrew and Dr. Jim Buckley. "Reengineering Towards Components Using 'Reconn-exion'." *Proceedings of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software*

Engineering (FSE-13). Sepetember 5-9, 2005. Lisbon, Portugal.

(Lewerentz and Rust 2000). Lewerentz, Claus and Heinrich Rust. "Are software engineers true engineers?" *Annuals of Software Engineering* 10: 311-328. 2000.

(Li, et al. 2006) Li, Yan, Jian Wang, Xianglong Li, and Wu Zhao. "Design creativity in product innovation." *International Journal of Advanced Manufacturing Technology.* Springer-Verlag London Limited. 2006.

(Linger 2004) Linger, Richard C. "Can Aspects of Software Trustworthiness be made Computable?" *Proceedings of the Center for National Software Studies Workshop on Trustworthy Software* (NPS-CS-04-006). 2004.

(Littlewood et al. 1995) Littlewood, Bev, Martin Niel and Gary Ostrolenk. "The Role of Models in Managing the Uncertainty of Software-Intensive Systems." *Reliability Engineering and System Safety* 46. 1995. (Love 2002) Love, Terence. "Constructing a coherent cross-disciplinary body of theory about designing and designs: some philosophical issues." *Design Studies* 23(3): 345-361. 2002.

(Maier 1977) Maier, Manfred. *Basic principles of design.* Van Nostrand Reinhold: New York. 1977.

(McBride 2004) McBride, Matthew R. "The Software Architect: Essence, Intuition, and Guiding Principles." *OOPSLA '04.* Vancouver, British Columbia, Canada: October 24-28, 2004.

(Naur and Randell 1969) Naur, P. and B. Randell (Eds.). "Software Engineering: Report of a conference sponsored by the NATO Science Committee." Garmisch, Germany. October 7-11, 1968. 1969.

(Nguyen and Swatman 2005) Nguyen, Lemai and Paul A. Swatman. "Promoting and Supporting the Creative and Insight-Driven RE Process Using Design Rationale." Information Systems Laboratory at University of South Australia. 2005.

(NIST 2002) National Institute of Standards & Technology. "Planning Report 02-3: The Economic Impacts of Inadequate Infrastructure for Software Testing." May 2002.

(NSF 05-620) National Science Foundation Program Solicitation: "Science of Design (SoD) – Software-Intensive Systems." NSF 05-620.

(Perry and Wolf 1992) Perry, Dewayne E. and Alexander L. Wolf. "Foundations for the Study of Software Architecture." *ACM SIGSOFT: Software Engineering Notes* 17(4): 40-52. 1992.

(Petroski 1994) Petroski, Henry. *Design paradigms: case histories of error and judgment in engineering.* Cambridge University Press. 1994.

(Putnam 2000) Putnam, Janis R. *Architecting with RM-ODP.* Prentice Hall PTR. 2000. p. 25.

(Robillard 2005) Robillard, Martin P. "Automatic Generation of Suggestions for Program Investigation." *Proceedings of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13).* Sepetember 5-9, 2005. Lisbon, Portugal.

(Sewell and Sewell 2002) Sewell, Marc T. and Laura M. Sewell. *The software architect's profession:*

an introduction. Prentice Hall PTR: Upper Saddle River, NJ. 2002.

(Shaw 1994) Shaw, Mary. “Patterns for Software Architectures.” Submitted to *First Annual Conference on the Pattern Languages of Programming.* 1994.

(Shaw and Garlan 1996) Shaw, Mary and David Garlan. *Software architecture: perspectives on an emerging discipline.* Prentice Hall: Upper Saddle River, NJ. 1996.

(Simon 1996) Simon, Herbert A. *The Sciences of the Artificial.* MIT Press. Third Edition, 1996.

(Skyttner 2001) Skyttner, Lars. *General systems theory: ideas & applications.* World Scientific: Singapore. 2001.

(Sommerville 1996) Sommerville, I. *Software Engineering*. Addison-Wesley. Fifth Edition, 1996.

(Thomson and Schmoldt 2001) Thomson, Alan J. and Daniel L. Schmoldt. “Ethics in computer software design and development.” *Computers and Electronics in Agriculture* 30: 85-102. 2001.

(Vidal, et al. 2004) Vidal, Rosario, Elena Mulet, and Eliseo Gómez-Senent. “Effectiveness of the means of expression in creative problem-solving in design groups.” *Journal of Engineering Design* 15(3): 285-298. 2004.

(Vahidov 2006) Vahidov, Rustam. “Design Researcher's IS Artifact: a Representational Framework.” *DESRIST 2006 Proceedings.* February 24-25, 2006. Claremont, CA.

(Vidosic 1969) Vidosic, Joseph P. *Elements of Design Engineering.* The Ronald Press Company: New York. 1969. p. 16.

(Weinberg 1975) Weinberg, Gerald M. *An introduction to general systems thinking.* Wiley: New York. 1975.

(Weinberg 1982) Weinberg, Gerald M. *Rethinking systems analysis and design.* Little, Brown: Boston. 1982.

(Willem 1990) Willem, Raymond A. “Design and science.” *Design Studies* 11(1): 43-47. 1990.

(Yang and Mei 2006) Yang, F. and H. Mei. “Development of software-engineering: Co-operative efforts from academia, government and industry.” In *Proceedings of the 28th International Conference on Software Engineering.* May 20-28, 2006. Shanghai, China.

(Yoshioka et al. 1993) Yoshioka, M., M. Nakamura, T. Tomiyama, and H. Yoshikawa. “A design process model with multiple design object models.” In *Design Theory and Methodology – DTM ’93*: 7-14. 1993.