

A manifesto for cooperative human / machine interaction in object-oriented conceptual software design

Advanced Computation in Design and Decision Making group Technical Report TR091006

C. L. Simons, I. C. Parmee,
Advanced Computation in Design and Decision Making,
Faculty of Computing, Engineering and Mathematical Sciences,
University of the West of England,
Bristol, BS16 1QY, UK
+44.117.3283135, +44.117.3283137
{chris.simons,ian.parmee}@uwe.ac.uk

Abstract

Conceptual software design presents many challenges to practitioners and yet is crucial to the success of software development. However, current computational tool support for conceptual software design is poor. One possible cause for this is that current computational tool support typically fails to exploit the natural cognitive design behaviours observed in software engineers. This manifesto declares that future computational support for conceptual software designers should exploit not only the specific strengths of designers and machines, but also their interaction during conceptual design. In addition, this manifesto builds on empirical evidence of cognitive behaviours of software engineers to advocate that man / machine interaction should be cooperative rather than competitive. Cooperative interaction presents significant opportunities for computational support for the discovery of substantive design knowledge and implicit learning in designers and machines alike.

1. Conceptual object-oriented software design

The process of conceptual object-oriented software design involves analysis of the problem domain to derive important and relevant concepts which are designed as classes. Such classes become an important part of the specification of the software system-to-be, and are subsequently implemented in a programming language to produce an executable software program. The designer's synthesis of these conceptual design classes is critical to development; such classes ensure that crucial concepts of the software problem domain drive the implementation. However, there exists a body of empirical evidence to suggest that conceptual object-oriented software design is difficult for human software designers to perform and learn. Studies by Guindon (1990a, 1990b) have shown that knowledge of design problem domain is difficult to grasp, especially when the problem domain is novel and

uncertain to the designer. Guindon also observes that specification of the software design problem may be incomplete and ambiguous, and that designers typically infer additional design requirements as their knowledge and understanding of the design problem improves. Formulation of conceptual software design solutions is also problematic according to Guindon, as evaluation criteria of the design solution may be poorly defined, and no pre-determined design path from design problem to solution exists. Glass (2003) suggests that initial software design solutions are likely to be wrong and seldom optimal. In addition, difficulties are also encountered by designers when learning and gaining experience of how to formulate conceptual design solutions. Svetinovic (2005) observes that some undergraduate students “*just don’t get it*” when trying to identify concepts via classes in object-oriented domain analysis. According to Books (2005), among experienced practitioners, software developer performance varies from more than 5:1 from the best developers to the worst. Furthermore, comprehension of conceptual software designs is subjective. Wirfs-Brock (2006) points out that that evaluation of conceptual software design classes is difficult within a team, as rarely do different designers agree on what makes good classes in a conceptual design. She further observes that finding the right level of abstraction in conceptual classes takes significant practice and experience. The potential impact of these difficulties is difficult to judge, but poor conceptual class designs may have significant untoward downstream consequences for software development. A previous literature survey by Simons *et al.* (2003) suggests a mixed picture of success for conceptual software design.

2. On the challenges of conceptual software design

Simon (1996) proposes a “sciences of the artificial”, encompassing the “*science of design*”. While describing the science of design, Simon describes how the design process might best accommodate the design environment’s constraints, including both the designer’s own cognitive and social constraints. Within such a science of design, Simon (1973) describes the nature of design problems as “ill-structured”, wherein there exist incomplete and ambiguous specification of design goals, no predetermined solution path, and the need to integrate many areas of knowledge. Coping with these characteristics while designing for ill-structured design problems is an inherently non-trivial task, especially when the design problem is novel and uncertain.

Exploring the areas of a software designer’s cognitive and social constraints, a number of software design practitioners (e.g. Beck 2000, Cockburn 2002, Schwaber and Beedle 2002) have emphasised the human aspects of attempting software design. For example, Cockburn characterises software design and development as a “*cooperative game of invention and communication*” and illustrates this with two analogies of software design. The first analogy requires a team of poets to produce an epic poem: each poet is responsible for producing component verses that must be integrated to produce the final poem. The problem domain of the poem is ill-structured yet numerous individual conceptions of human experience must be captured in verse and combined to arrive at a coherent whole. The second analogy is a team of rock climbers, where the success of the climb

depends upon all team members inventing their climbing tasks which, when communicated and synchronised, achieve the shared goal of the team reaching the summit. Underpinning both analogies is the invention of abstract solution ideas for ill-structured design problems and their subsequent communication among a cooperating team. Cockburn speculates that “*managing the incompleteness of communications*” is core to mastering software development.

Cockburn goes so far as to suggest that perfect communication is impossible. A possible explanation to support this is provided by Sperber and Wilson (1995), who point out that communication involves both messages and signals. A message is a representation internal to the communicating device (i.e. person); a signal is a modification of the external environment which can be produced by one device and recognised by the other. In verbal communication, the signal is an utterance; in written or pictorial communication, the signal is a symbol. In either case, the sender is responsible to mapping the message to a signal and the recipient is required to “unmap” the signal to the desired message. Sperber and Wilson describe two theories to explain the mapping / unmapping process; one suggests encoding, the other inference. Encoding may be the prime mechanism in well understood situations, but in unknown contexts, inference is dominant. Sperber and Wilson go on to suggest that perfect inference requires the mutual knowledge of both parties to the communication context which may go some way to explain Cockburn’s assertion on the impossibility of perfect communication.

However, in spite of this, many software designers do achieve elegant conceptual software designs. In her much-cited empirical study, Guindon (1990a) provides some remarkable insights. In the study, 3 designers of varying expertise are observed as they attempt to solve a standard software design problem, the Lift Control Problem. As design proceeds, Guindon notes the following design activities:

- mental simulation of scenarios from the problem domain, which led to
- understanding and elaboration of the problem specification, including inference of further requirements and constraints, and
- generating candidate design solutions, including their representation, the addition of new partial solutions, and their mental simulation and evaluation.

Guindon observes that the designers continually adjust and iterate over various levels of abstraction and differing viewpoints on both the problem and generated solutions, thus the design process did not sequentially flow from problem to solution. In addition, at various times in the design, designers express “Ahaa!” moments, where the sudden discovery of partial solutions or requirements emerges. Guindon also notes that one moment of sudden discovery might often lead to another in a short space of time.

It is useful at this point to speculate on the causes of the cognitive behaviours of the software designers that might in some way account for the highly iterative nature of the software design

process and the sudden discovery of partial solutions or requirements. However, it is not the intention of this technical report to provide a comprehensive survey of the cognitive psychology of conceptual software design. Rather, this section now conjectures on the causes of these interesting cognitive design behaviours with the intent that they may prove to be fruitful drivers for requirements for interactive tool support for conceptual software design provided later in this report.

Guindon (1990a) suggests that one cause may be the opportunistic behaviour of designers. She observes that the designer's rate of progress of solving a design problem is not linear but proceeds with spasmodic opportunistic leaps associated with sudden emergence of new knowledge about the design problem. Such opportunistic discoveries typically result in the generation of partial design solutions that are followed by unplanned but drastic changes in design activities. Guindon also emphasises discovery of new design problem knowledge by inference and notes that this results in immediate solution generation by the designer. Such inferences appear to be best stimulated by mental simulations of design problem scenarios. In this context, an inference may be considered to be the act or process of deriving a conclusion based solely on what is already known. Sperber and Wilson (1995) explain the phenomenon of sudden knowledge discovery with the notion of relevance. Sperber and Wilson consider the efficiency of short-term human cognition, in which great cognitive processing may be achieved in a short space of time. They distinguish between old knowledge and new knowledge emerging from a problem situation. Some new information may be unconnected with anything of the person's representation of the problem at hand, resulting in heavy cognitive processing to derive any useful new knowledge. However, other new information may be connected with the old. According to Sperber and Wilson, "*when these pieces of new and old items of information are used together as premises in an inference process, further new information can be derived: information which could not be inferred without the combination of new and old premises. When the processing of new information gives rise to such a multiplicative effect, we call it relevant. The greater the multiplication effect, the greater the relevance*". It seems logical to assume that designer's sudden leaps of knowledge discovery may, at least in part, involve the multiplication of design knowledge via inference based upon relevant items of information emerging from mental simulations of the design problem domain and generated design solutions. It also seems logical to assume that the notion of human cognitive design information processing efficiency depends upon (1) the relevance of the new information emerging from the design domain, and (2) the cognitive abilities of the designer to infer.

A further possible explanation for the high cognitive processing efficiency observed in these sudden leaps in knowledge discovery is offered by Gigerenzer *et al.* (1999), who describe the use of fast and frugal heuristics. Gigerenzer *et al.* suggest that where there is sufficient information available in a problem situation, the human mind solves the problem in a systematic, rational and logical way in which the person is able to articulate clearly the steps involved with the process. However, in problem situations where insufficient information is available for systematic analysis, human minds may resort

to the use of fast and frugal heuristics. Such heuristics are rules of thumb that solve a problem by performing a limited search through objects or “cues” without any attempt to optimise. (A cue is a feature of something perceived that is used in the brain’s interpretation of the perception). A key characteristic of heuristics (and indeed inference) is that the human mind cannot articulate how the generated solution has been arrived at – it just happens. Gigerenzer *et al.* offer a number of examples of fast and frugal heuristic use. Firstly they describe the recognition heuristic. In a study, a number of undergraduate students are asked to rank a number of German cities according to population size. Evidently, the names of some of the cities are unknown to the students. However, if the names of two German cities are placed in a tournament and one of the names is recognised by the student, the recognised name wins and is deemed larger with respect to population size. Thus each name competes in tournament with all other names, and a ranking score emerges. Gigerenzer *et al.* report a close correspondence between the actual sizes of the cities, and the ranking achieved by the recognition heuristic. A second use of the recognition heuristic is classification. For example, if one wished to identify a bird seen flying in the sky, a feature of the bird (e.g. hooked beak) may be observed. All those birds that do not possess that feature are eliminated as possibilities. A further feature of the bird (e.g. grasping claw) is then observed, and all birds without this feature are eliminated. The step is repeated for further features until only one possibility remains. Using feature recognition and elimination, classification of an entity is cognitively fast and frugal.

Clearly, there are limitations to the recognition heuristic. Firstly, its use is heavily dependent on the problem context; what works in environment does not work in another. Secondly, successful use of the heuristic depends upon the human mind having partial knowledge of the problem context. For instance, if the mind has no knowledge of the problem context, recognition is not possible. Further, if the mind has total knowledge of the problem context, everything is recognised. Only when the mind possesses partial knowledge of the problem context is the heuristic possible. However, partial knowledge is the typical situation found in ill-structured software design problems. For software designers with at least some previous design experience or access to design principles or patterns, the recognition heuristic may help to explain sudden discoveries of new knowledge. Gigerenzer *et al.* define the recognition heuristic as “*if one of the two objects is recognised and the other is not, then infer that the recognised object has the higher value*”. Such a heuristic appears consistent with the multiplicative effect observed in inference, where the human ability to recognise some objects in situations of partial knowledge quickly leads the designer to relevant information.

Other authors have also speculated on the causes of software designers’ cognitive behaviours that might account for the sudden discovery of partial solutions or requirements. For example, Parmee and Abraham (2004) cite Westcott’s insights into intuition. Westcott (1968) builds on early work by Jung (1926) who described sensation and intuition as irrational mental functions i.e. they involve no rational judgements. Westcott cites Jung’s description of intuition as “*...the process of perceiving, immediately, and unconsciously, the possibilities and potentialities of the objects which are the focus*

of attention, whether internal or external [to the mind]". Westcott then offers his own definition of intuition as "...the process of reaching a conclusion on the basis of little information which is normally reached on the basis of significantly more information" and notes that for a conclusion to qualify as intuitive, the thinker must not know how they reached the conclusion, suggesting that unconscious use of cues in the problem environment are key. Following empirical observation, Westcott characterises people as belonging to one of four groups according to their intuitive behaviour:

- "Successful intuitives" – those who explore uncertainties and entertain doubts far more than other groups do, and live with these uncertainties without fear. They enjoy taking risks, and are willing to expose themselves to criticism and challenge.
- "Wild guessers" – those who appear to be striving for a grasp of reality which so far eludes them, and they are likely to attempt different modes of attack in a somewhat chaotic manner.
- "Successful problem solvers" – those who tend to have a very strong preference for order, certainty, and control. This subject is conservative, cautious, somewhat repressed, and they function well in situations where expectations are well managed and well met.
- "Cautious careful failures" – their world view seems to be in which everything is risky at best, and they essentially powerless to influence or control. They are quite conservative, presumably, as the best defence against the great uncertainties of life, and they seem to wander through life just keeping their heads above water, not making waves.

Westcott then poses the question of where the behaviours of "successful intuitives" may be useful. He suggests that *"we must look to situations in which information, explicitness and redundancy, are just not available. One such situation is at the frontiers of knowledge in any field. At these frontiers, we might well expect that the ability to solve problems on minimal information is the only way that problems can be solved at all. As long as detail, redundancy and explicit information are necessary to an individual, he will be incapable of solving these problems; he will be incapable of going on beyond frontiers. The "major breakthrough" can be seen as the solution to a problem for which apparently only insufficient information is available."* The notion of successful intuitive thinking leading to a "major breakthrough" also appears consistent with the sudden discovery of new knowledge in a design.

One possible explanation for sudden knowledge discovery is offered by Dominowski (1995) who points out that one aspect of general problem solving involves knowledge of the connections between various components of a problem. He describes "productive" problem solving where the discovery of new connections and relations between components gives rise to "insight". However, Dominowski also suggests that "productive" problem solving is achieved when a person's attention is directed to the appearance of new connections between problem components, implying that the emergence of new connections depends on contextual influences. This explanation is consistent with

Schooler and Melcher (1995) who propose that perceptual restructuring of connections and relations between components of a problem via the presentation of fresh perspectives enables intuitive creativity. In one investigation into perceptual restructuring of an engineering design problem, Parmee and Abraham (2004) explore the manner in which designer understanding of dependant engineering design spaces can be supported by graphical design representations of complex variable / objective relationships from a variety of perspectives. Results of high-dimension multi-objective searches are first clustered to identify regions of high performance in design space, and then further selected by an adaptive filter into a final clustering set. Design variants in the final clustering set are presented to the user from a variety of perspectives to stimulate knowledge discovery. Such perspectives include Pareto-optimal fronts in orthogonal variable / objective spaces, two dimensional hyperplanes, and parallel co-ordinate box-plots.

Interestingly, both Dominowski (1995) and Smith (1995) point out that it is possible for a person to get “fixed” in a particular representation, or set of component connections, leading to the person “going around in circles” or “being stymied”. In such a situation, Dominowski and Smith suggest a period of “incubation” to allow the person to contribute to insights experiences, since the passage of time allows the connections between problem components in consciousness to fluctuate. Again this is consistent with Parmee and Abraham (2004) who suggest that insight and intuition may arise such incubation periods while designers reflect and “*surf what we already know*”. They also suggest that external stimulation from diverse problem perspectives is an effective tactic for breaking out of design fixation.

A further possible explanation for knowledge discovery is the phenomenon of implicit learning. Implicit learning is described by Berry (1997) as “*typically used to characterise those situations where a person learns about the structure of a fairly complex stimulus environment, without necessarily intending to do so, and in such a way that the resulting knowledge is difficult to express*”. Because it is difficult, if not impossible, for the individual to verbally express how the knowledge was acquired, Berry speculates as to extent to which implicit learning is abstract and unconscious, and perhaps ‘sensed’ from the experience of involvement in a design episode. While the exact cognitive mechanisms of implicit learning remain unclear, numerous researchers (see Berry, 1997) report the acquisition and retention of implicit knowledge in a variety of experimental situations, and it has been observed that efficient learning happens when there is a blend of explicit and implicit learning. Parmee and Abraham (2004) report one tangible example of conceptual engineering design in which direct support for implicit learning is attempted. In their case study, it is found that the visualisation of diverse perspectives on multi-object search results is essential to gaining knowledge of the overall complexities relating to multi-dimensional design spaces. It is interesting to speculate how such findings of support and stimulation of implicit learning might be applied to conceptual software design.

3. Current tool support for conceptual software design

Current industrial tool support for object-oriented conceptual software design typically provides support for the later stages of design after choices concerning the identification and purpose of candidate classes have been made by the human designer. A plethora of commercial modelling tools is currently available, many of which comply with the industry standard Unified Modelling Language (UML) object-oriented notation. Comprehensive lists of such UML tools are available from the Object Management Group (OMG, 2006).

Current research initiatives into tool support for conceptual software design can be grouped into a number of categories: tools that use natural language processing (NLP), tools that use search based search approaches, and tools that in some way other support the design process. Each category is briefly surveyed as follows.

A variety of tools have been researched that exploit natural language processing (NLP) e.g. Larvet (2002), Mich (2002), Harmain (2003), Liu (2004). Tools adopting this approach examine the text of software design problem and requirements documents, then parse and analyse the text for identifiers depending on their grammatical context. For example, nouns are typically extracted and put forward as candidate concepts and presented to the designer as classes. Efforts are made to investigate conceptual semantics, for instance to isolate synonyms. However, it is characteristic of all NLP approaches that the quality of the classes arrived at depends greatly on the quality of the software design problem documents. Given the ill-structured nature of software design problems and the difficulties of articulating software problems described in the previous section, this appears to be a significant limitation. Furthermore, conceptual design solutions arrived at are confined to the ways in which NLP converges on concepts; no attempts are made to explore a range of solution possibilities.

A number of tools have been researched that investigate the possibilities of search-based approaches. After Harman and Jones suggested the application of search to the software engineering life cycle in 2001, the use of search techniques for software clustering, modularisation and refactoring has been widely investigated. Recent examples can be found at Mitchell and Mancoridis (2006) and Seng *et al.* (2006). However, in terms of the software development lifecycle, such clustering and refactoring algorithms are applied downstream after the software engineer has already designed and implemented the software system. This differs significantly from the aims of this technical report where tool support for upstream conceptual software design is the goal. On a different tack, Lo and Chang (2004) report the application of clustering techniques to software component architectural design, but require the human design to have produced requirements and conceptual class designs beforehand. The potential for designer interaction with the search is not discussed. Thus in common with the search approaches described above, there exists a certain detachment between the designer and the computational search. Such approaches rely upon on cognitive design behaviours of the

human designers generating (at least initial) conceptual designs *before* the search techniques can be applied.

Interesting research efforts have also been made to computationally support the designer during the design process in various ways. For example, in 1992, Guindon took account of the opportunistic cognitive design behaviours and produced a set to requirements for an “intelligent design assistant” that would afford opportunities for sudden design insights and knowledge discovery. Unfortunately, it appears that the development and application of such a design assistant in practice have not been reported subsequently in the research literature. More recently, Egyed (2006) reports an interesting support tool that applies the use of constraint satisfaction problem solution techniques (or CSTs) during software design. After the human designer has sketched out an initial, incomplete conceptual class design, the tool extrapolates all feasible solution possibilities to complete the design using UML design syntax and well-formedness rules. Using this tool, the designer may thus interact with the tool to eliminate extrapolated design possibilities of poor fitness. The tool also appears useful in allowing dynamic simulations of incomplete conceptual designs, stimulating the designer to acquire further knowledge with which further design extrapolations may be eliminated. Egyed reports encouraging results for a variety of industrial design situations. However, the human designer is still required to submit initial partial conceptual designs before machine extrapolation is possible.

Although beyond the field of software design, it is noteworthy that a large body of research into computational support for conceptual design is found in engineering design. For example, the capture of conceptual sketches in architectural design is reported by Do (2002), in which steps towards a computational sketching interface are described. An example of computational support for collaborative conceptual design is described by Wang *et al.* (2002), who survey the state-of-the-art and future trends of distributed computation support environments for interactive conceptual design in engineering and architectural design. In addition, much research effort has been conducted into Interactive Evolutionary Design Systems (IEDS) in the field of engineering design by Parmee and Cvetkovic (2000), Parmee (2002), Parmee and Abraham (2004) and more recently Machwe and Parmee (2005) with promising results. A principal goal of such Interactive Evolutionary Design Systems in engineering design is precisely to afford opportunities for the designer to achieve moments of sudden knowledge capture, insight and creativity, and implicit learning in ill-structured design problems.

It is interesting to note that both industrial tool support and research into computational support for conceptual software design remains reliant on the human designer at least partially (if not totally) solve the design problem manually before computational tools can “support” the designer. In one sense this is puzzling, as some time ago Guindon (1992) suggested computational design support tools should be compatible with and exploit the cognitive behaviours of human designers. This suggestion has been echoed by Robillard (1999) who points out that software engineers have made little use of the empirical findings of the cognitive scientists who have observed the design behaviours

of software engineers and the design knowledge that is discovered. We therefore suggest that computational support tool for the human designer in conceptual software design should take account of the software designer's cognitive design behaviours and exploit them to the full.

4. Turing test revisited

In his seminal paper reflecting on computing machinery and intelligence, Alan Turing (1950) suggested that instead of asking whether machines can think, we should rather ask if a machine could pass a behavioural intelligence test – the so-called “Turing test”. Turing refers to this test as an “imitation” game, in which a human interrogator converses (via typed messages) for 5 minutes with either a computer program or a person located in a different room. The interrogator then must guess if the conversation is with a human or a program; the program passes the test if the interrogator is fooled 30% of the time. Turing believed that by the end of the twentieth century, computational machines would exist that could pass the test, and he concludes his paper with the hope that “*machines will eventually compete with men in all purely intellectual fields*”.

Since 1950, computational machines have come to perform tasks in a wide variety of industrial fields as well or better than humans. In research domains, the notion of human-competitive machine programming has proved a source of inspiration and various human behaviours have been taken as a benchmark against which computer programs are supposed to out-perform. And yet there remain many open-ended tasks that computational machines do not yet excel, including Turing's task of two people carrying on a conversation.

As well as machines *competing* with men (as in Turing's terms), might not machines *cooperate* with men? As differences between human mental processes and computational processes in open-ended tasks such as conceptual design are manifest, it seems logical to exploit both the strengths machines and people in the course of their interaction. Perhaps because of the competitive nature of the “imitation game”, the significance of the interrogator's interaction with the machine is not discussed by Turing. Possibly if the goal of an imitation game were to be restructured as a cooperative game, for instance by the cooperative solving of a crossword puzzle (via typed messages), the interaction between person and machine would be highly significant.

It is important to note that the interaction between people and machines in a cooperative imitation game is different to what might usually be considered to be human-computer interaction (HCI). In the majority of industrial computer programs, HCI is typically concerned with the effectiveness and usability of the computer program by a variety of people, but the computer program is largely un-reactive and unchanging in its substantive dealings with people. Only people's behaviour, in the course of interacting with the computer, significantly changes. But according to the OED, to interact is to “*act in such a way as to have an effect on each other*”. Wikipedia goes further to define an interaction as “*a kind of action that occurs as two or more objects have an effect on each other. The idea of a two-way effect is essential to the concept of interaction, as opposed to a one way*”.

causal effect. Combinations of many simple interactions can lead to surprising emergent phenomena". If a computer program and a person are to interact in the sense of the above definitions, then both the person's behaviour and the computer program's behaviour should change in the course of the interaction. Put another way, both the person and the computer program should learn from the interaction and adapt their behaviours to better cooperatively achieve the goal of the task in hand. This cooperative interaction would ideally be an integral component of tool support for conceptual software design - a cooperative game of invention and communication among software designers.

In his paper, Turing anticipates a number of potential objections to his proposals, including the assertion that a computer machine "*can never do anything new*". Turing counters at some length with a discussion about "*learning machines*" and puts forward the conjecture that at some point in the future, a computer program may be taught new behaviours much in the same way as a child may be taught at school. While Turing's insight is interesting, in the present day we recognise that as well as training programs by supervised learning, a variety of other mechanisms exist for machine learning, not least unsupervised learning, reinforcement learning, case-based reasoning and evolutionary approaches. However, Turing does not speculate how learning might occur by either person or machine in the course of their interaction during the imitation game.

One conjecture of Turing is intriguing. As a mechanism to promote machine learning, Turing speculates on the possibility of "injecting" an idea into the machine. He imagines a machine that might simulate the human mind. Ideas are then "injected" into this human machine. Turing suggests that frequently an idea may provoke a limited response, but a smallish proportion of ideas are "super-critical". Turing suggests that "*a super-critical idea presented to such a mind may give rise to a whole 'theory' consisting of secondary, tertiary and more remote ideas*". We suggest that Turing's conjecture of super-critical idea "injection" is consistent with the cognitive phenomenon of sudden knowledge discovery described in the previous section of this report.

We speculate that if the Turing test imitation game were to be reformulated for conceptual software design, it may consist of a team of software designers each communicating remotely to cooperatively invent software design, but, unknown to others, one of the team is a computing machine. The interaction of the designers and the machine would enable both to discover knowledge about the design problem to cooperatively complete the design solutions. Both human and machine designers would iteratively learn from each other's behaviour in a cyclical loop as previously described by Parmee (2005). At the end of a 5 minute design session, the object of the game is for the human designers to agree upon which of the other designers is a computing machine. The machine wins if it can fool the human for more than 30% of the time.

While not exactly trying to construct a machine to pass such a test, the next section uses these ideas and the cognitive behaviours described in the previous section to sketch out pragmatic requirements of a prototype interactive conceptual object-oriented software design environment.

5. Requirements for tool support with man / machine interaction

Using the cognitive behaviours of conceptual software designers described in section 3 and the proposal for cooperative man / machine interaction as motivating factors, we speculate that the following list of features would likely be present in an Interactive Conceptual Object-Oriented Design Environment (ICODE), in no particular order of priority. Of necessity, such features tend more to vision than detail as they form the basis for future ‘proof of concept’ research and experimentation.

Feature: Software Design as Search

Designers, especially novice designers, find it difficult to synthesis initial partial solutions to the specified design problem. Designers also find it difficult to formulate a range of conceptual design solution variants and subsequently evaluate the variants according to objective design principles and metrics. To alleviate this difficulty, designers may exploit a computational search engine using stochastic search and exploration techniques offered by evolutionary algorithms within the Interactive Conceptual Object-oriented Environment. Previous work by the authors (Simons and Parmee 2006a, 2006b) reports on multi-objective evolutionary algorithms using class coupling, cohesion and size as objective fitness functions. Such multi-objective evolutionary algorithms produce useful and interesting design solution variants of equivalent or superior fitness to those produced manually by the human designer. Crucially, support offered to designers using such an evolutionary search engine is twofold:

1. the problem and solution representation enable the automatic generation of a population of initial design solution variants which greatly assists the designer in getting started, and
2. the evolution of the population of design variants steers the population towards useful and interesting candidate solutions of superior fitness.

Simons and Parmee report that the variety and number of design solution variants generated and evolved greatly exceeds the number and variety that a human designer typically manipulates during design. Thus the designer’s knowledge of the design may be stimulated with a range of original possibilities to promote knowledge discovery.

Feature: Visualisation and Recognition

Guindon (1992) and Parmee (2002) report that visual representations of design solutions play a critical role in early design, especially in the formulation of solution concepts. The industry standard notation for object-oriented software modelling is the Unified Modelling Language (UML), thus within the Interactive Conceptual Object-oriented Design Environment (ICODE) designers may inspect visual representations of design concepts as depictions consistent with the UML. Furthermore, the designer may set filter criteria relating to various objective fitness functions, to home in on a range

of design solutions above a certain fitness threshold, or various hyperplanes relating to selected fitness functions.

As selected designs of superior fitness are visualised, designers may exploit recognition heuristics according to “cues” (or features) immediately apparent. For instance, if no obvious state changing methods are evident in a candidate class in a design variant, it may be designated an <<information holder>> stereotype. Alternatively, if the majority of methods in a class appear to change the internal state of the candidate class, it may be designated with a <<control>> stereotype. (For a detailed description of stereotypes, the reader is directed to Wirfs-Brock and McKean, 2003). Designers may then express and manipulate certain trade-off preferences, e.g. centralised versus distributed designs. In such a trade-off, for example, the search engine may steer the search away from a single, large <<control>> class (a so-called ‘God-like’ class) to many smaller, distributed <<control>> classes. To stimulate further knowledge discovery and insight, the designer may use further cues and emerging class stereotypes as the basis of “perspective” recognition heuristics. As suggested by Wirfs-Brock and McKean (2003), designers recognise partial design solutions according the perspectives such as “*representations of real-world things the software needs to know about*”, “*information that flows through the software*”, “*decision making, control and coordination activities*” etc. Using fast and frugal recognition heuristics, the designer may visualise partial or whole design solutions via perspectives. The value of such perspectives is that, especially for expert designers, a more abstract viewpoint may be adopted to reveal design insights and knowledge relating to design patterns (Gamma *et al.* 1995, Fowler 1997, Arlow and Neustadt 2004) or problem frames (Jackson 1995).

Alexander (1964), while discussing the “goodness of fit” of a solution to a design problem, notes that designers are instinctively more adept at recognising an unfit solution rather than a fit solution. However, the designer may be stimulated by possible insights that even a poor-fitting design solution may offer, particularly if the insight is relevant (as in Sperber and Wilson’s (1995) terms). Thus designers may, from time to time, “inject” design ideas and knowledge into the population of design solutions to stimulate design knowledge discovery in two ways:

- inject “what if?” ideas in the partial class design solutions to see if any “cues” emerge which may subsequently be exploited by the recognition heuristic, and
- dynamically simulate scenarios of execution derived from knowledge of the problem domain use cases. Such scenario simulations use potential instances of domain concepts as classes, and simulate their dynamic interaction to match the desired behaviour of use cases. Dynamic simulation of scenarios greatly improves knowledge of design problem and solution traceability by providing the opportunity to disambiguate any inferred knowledge and cross-validate the dynamic behaviour of the simulation against the static representation of the problem domain concepts visualised as classes.

Feature: Progressive Aggregation

Designers may find it difficult to impose structure and architecture on the initial design solution, particularly at the early stages of design when the design problem is ill-structured, and knowledge of the design problem and solution is partial and incomplete. As a way forward, designers mentally isolate partial components of the design problem, and synthesise corresponding partial design solutions. As design proceeds, both the interfaces between partial design solutions and the constituent classes within the partial solutions are expanded upon and the integration of the partial solutions validated against design knowledge obtained from mental simulations of design scenarios. ICODE should support the formulation and visualisation of such partial design solutions (sometimes referred to as “subsystems” or “components”), building upon the UML notation of the package. Numerous clustering algorithms have been reported in the software development cycle (e.g. Lo and Chang 2004, Harman *et al.* 2005) and may be exploited to provide support for automated progressive clustering or aggregation of classes into packages. The designer may be presented with multiple package compositions to provide relevant knowledge with which to make informed and insightful architectural design trade-off decisions.

Feature: Problem Reformulation

Guindon (1990) highlights the difficulty of “inferred” requirements that arise in the problem domain as the design problem becomes increasingly understood by the designer. Glass (2003) echoes this observation and suggests that “*when moving from requirements to design, there is an explosion of ‘derived requirements’ caused by the complexity of the solution process*”. Designers typically iterate over representations of both the design problem and solution to validate and inform their understanding of both. Indeed reformulating the design problem making it amenable to solution is frequently a necessary and integral part of the design process. In the field of engineering design, Parmee (2002) reports an Interactive Evolutionary Design System that concentrates on background human-computer interaction relating to the machine-based generation of high-quality information, that when visually presented to the designer, supports a better understanding of the problem domain. ICODE should thus support the iterative reformulation of the problem statement, while maintaining consistency and traceability between the problem and solution representations.

Feature: Design Elegance

In the inherently multi-objective process of software design, fitness functions may be both quantitative objective fitness functions to assess the structural integrity of designs (e.g. class coupling and cohesion) and human designer intuitive notions of elegance. Although sometimes overlooked by software design methodologies, producing elegant designs can be very important, not least to a software designer’s perceptions of their own worth. Furthermore, software practitioners who advocate

placing people at the heart of the software design process (e.g. Beck 2000, Cockburn 2002) place great emphasis on elegance as simplicity of design.

Gelernter (1998) describes the notion of “machine beauty” and explains that “*the beauty of a machine lies in a happy marriage of simplicity and power – power meaning the ability to accomplish a wide range of tasks...*”. Gelernter goes on to state that beauty is crucial to software design and development, and cites the recursive ‘quick-sort’ sorting algorithm as a thing of beauty. He compares the ‘quick-sort’ algorithm with other brute force sorting algorithms such as bubble sort, and notes that as well as being smaller and simpler in terms of algorithmic expression, it also significantly outperforms its non-recursive variant. Gelernter’s notion of beauty is also consistent with the heuristic maxim of Occam’s razor which advocates succinctness and economy in the expression of scientific theories.

To achieve elegance in software designs, practitioners frequently deploy “design guidelines”. Such documents provide designers with tactics for elegant designs, not only for the personal satisfaction of the designer, but also because elegant designs are better understood and communicated to other designers within the design team. Software design, as a cooperative game of invention and communication, is more efficient when designs are elegant and so more easily communicated and understood among team members. Because of this, ICODE should gauge elegance as powerful behaviour for simple classes, filter design solution variants from the search population by elegance for presentation to the designer, and stimulate the human designer to reflect upon elegance by presenting computationally intelligent design guidelines for elegance. This is consistent with the practice of “refactoring” in Extreme Programming (Beck, 2000). Beck urges designers to produce “*the simplest thing that could possibly work*”, adding that as well as solving the design problem, there should be no duplication in the design, and generally the smallest possible number of classes and methods. Exemplars of elegant conceptual designs are found in design patterns such as Fowler (1997) and Arlow and Neustadt (2004). (It may be highly productive in terms of sudden design knowledge discovery for ICODE to inject design patterns as partial solutions to the problem, as described with the Visualisation and Recognition feature above).

Feature: Episode Tracing

In the field of engineering design and architecture, the notion of a design “episode” is well understood. Lawson (2004, p17-18) describes the knowledge associated with architectural design and explains that the components of design thought appear to be made up of design events and design episodes. Lawson suggests that design events are the smallest indivisible components of design and may include “*a request for information, a structuring of a problem, a proposition about a possible solution characteristic, a representation of a solution characteristic, an evaluation of a solution characteristic. They may also include process intentions such as a reflection on the way a process is going, a decision to change direction, an evaluation of time or effort either past or future, and so on.*”

In a hierarchical way, Lawson also suggests that a group of design events may be closely related to each other, carried out to move the design forward in some way. It seems likely that once a focused designer embarks upon an episode, they are unlikely to quit until the purpose of the task related the episode is achieved.

Design episodes have been used by researchers as the unit of design on which analogical reasoning and case-based reasoning can be modelled and exploited. For example, Goel (1997) suggests that analogical reasoning appear to play a key role in creative design. The suggestion is based on the recognition and analogical transfer of design knowledge from a design episode with one design problem context to another. In a similar approach, Champin (2001) proposes a model and interactive engineering design support tool for case-based reasoning based on design episodes and events. Champin suggests the use of design episodes as the case bases, indicating that most of the knowledge used by the designer during an episode is implicit. Champin's interactive support tool enables the reuse of cases across multiple design episodes and problem contexts by the use of "adaptation episodes" which exploit the implicit knowledge contained within the episode.

ICODE should capture the interaction of design events and design episodes as mechanism for episode tracing, with the ability for the designer to stop the episode at any event for human reflection, or adjustment of preferences, filter settings, number of generations to progress the evolutionary search engine etc. The designer may then restart the episode. As the design episode proceeds, ICODE records design decisions made at significant episode cross-roads, preferences, filter parameters and criticality weightings as well as offering opportunities for the designer to make notes and fill "to do" lists. ICODE should also be aware of designer behaviours during an episode to exploit implicit learning mechanisms such as case-based reasoning.

Feature: Implicit Learning in cooperative man / machine interaction

The proposal put forward in the previous section is for cooperative interaction between man and machine in the game of invention and communication that is conceptual software design. The proposal also suggested that man and machine play to each other's strengths. However, it is envisaged that any interactive computational design environment should promote implicit learning on the part of *both* machine and man. Thus ICODE should promote the implicit learning of both machine and man, promoting synergy of mutual implicit learning.

Feature: Multi-user Collaboration – a cooperative team

As conceptual software design is typically a cooperative game of invention and communication, the interactive design environment for conceptual software design should enable multi-user design-time interactive collaboration, both between designers and between the design environment and designers. Underlying technologies are widely available for this purpose and research proposals for distributed, collaborative environments has been put forward e.g. Wu and Graham (2005).

Conjecture: An envisaged design episode

To illustrate how this might all come together, the following is a conjectured design scenario involving three human designers and the Interactive Conceptual Object-oriented Design Environment (ICODE):

1. Designer 1 commences a design episode within ICODE setting up preferences and priorities for the design problem under design.
2. Designer 1 uploads the design problem into ICODE. From the problem representation, ICODE generates an initial population of conceptual design solutions within the bounds of previously set parameters.
3. Designer 1 starts the evolutionary search engine with ICODE which evolves for 50 generations.
4. ICODE isolates and visualises the high performance design solution variants, presenting trade-off hyperplanes. ICODE also recognises from previous design problems that structural cues are evident and presents suggestions for candidate class stereotypes to Designer 1.
5. Designer 2 joins the design episode and verbally discusses the suggestions for stereotypes with Designer 1. Both designers recognise a trade-off between centralised and distributed control and agree that distributed control is preferred for this problem domain. Designer 1 adjusts ICODE fitness criteria accordingly.
6. Designer 1 restarts the evolutionary search engine with ICODE which evolves for 50 generations, and presents a visual representation of high performance design solution variants. ICODE recognises the potential for structural clustering in some design solutions of high fitness and suggests an architectural clustering for a subset of the design. Designer 2 disagrees with the ICODE suggestion, although on grounds of elegance and simplicity, manually isolates a similar but not identical cluster within a package. ICODE records this as a case of cluster trade-off preferences. Designer 1 and 2 verbally agree the cluster aggregation and ICODE suggests parallel evolutionary search for the partial cluster solution and the remainder of the design solution. Designer 1 and 2 discuss and accept the suggestion.
7. Designer 3 joins the design episode remotely and inspects the episode trace so far.
8. Designer 1 restarts the evolutionary search engine with ICODE which evolves for 50 generations, and presents a visual representation of the isolated cluster and the remaining high performance design solution variants.
9. Designers 1, 2 and 3 electronically discuss the visualisations. In the isolated cluster, they discover that the package of classes now shows good distribution of control. Designer 2 recognises a design pattern and injects the pattern via ICODE; ICODE records this and suggests that a further pattern may be applicable in the remainder of design solution, although Designer 3 disagrees, recognising different cues. Designers 1, 2 and 3 discuss this and based

on their improved understanding agree to further evolve the isolated cluster with similar preferences and apply the ICODE intelligent design guidelines for elegance and simplicity. However, for the remainder of the design, designers 1, 2, and 3 agree to use the ICODE Episode Tracer to roll back to the previous generation event, adjust the criticality of the preferences, and restart the evolution for 100 generations.

10. After 100 generations, the isolated cluster appears to have presented some interesting design variants of high performance that the designer could not have conceived of, stimulating insight and further understanding. However, only some of the design variants score highly with respect to elegance guidelines. Designers 1, 2, and 3 decide to follow their intuition to select a few design variants that trade-off performance with elegance. For the remainder of the design, ICODE has injected some unexpected ideas into the solutions, based on previous machine learning about successful design behaviours from previous design episodes and design problem contexts. New different cues emerge that are recognised independently by all three designers. Discussion between the designers connects the cues leading to the sudden discovery of two further clusters and their relationships. Within ICODE, the designers simulate some of the dynamic behaviours required by the use cases in the problem statement, and find that although one of these clusters appears satisfactory, the other is clearly inferior and difficult to understand.
11. ICODE records the design events and records the selections and preferences of each individual designer, comparing them with previous episode cases for potential learning. ICODE isolates the second satisfactory cluster, whereupon the designers agree to divide their efforts: designer 1 taking the first cluster, designer 2 the second and designer 3 the difficult remainder of the design.
12. While designer 1 and 2 successfully explore further elegance refactoring versus dynamic simulation trade-offs in ICODE, designer 3 is having less success. ICODE recognises ineffective design outcomes for the remainder of the design for a period of time, and so suggests a problem reformulation of the part of the problem that traces to the difficult remainder of the design.
13. After a further 50 generations of evolutionary search of the difficult remainder of the problem following problem reformulation, ICODE presents a visualisation of the high performance design wherein designer 3 recognises further cues for elegance and simplicity and so deems the high performance design satisfactory.
14. ICODE records the design episode as complete.

We suggest that such a design episode would be quicker than if the interaction were between human designers alone, as less delay would be experienced in formulation of design solution variants, the presentation of design solutions of superior fitness would be in real time and the communication

between designers is enhanced. We also speculate that the design solutions arrived at would be of superior fitness to those produce by human designers alone.

6. Conclusions

The notion that computational tool support for the early stages of software design should reflect the cognitive behaviours of designers is not new. Furthermore, in other fields such as engineering design and architecture, various interactive conceptual design environments have been investigated with promising results. However, to specifically address the challenges of conceptual software design, we suggest that computational tool support should exploit the cognitive strengths of designers, the processing power of machines, and their interaction. Such interaction should affect both designers and machines, and be cooperative rather than competitive. We further advocate that cooperative human machine interaction in conceptual object-oriented design affords significant opportunities for discovery of substantive design knowledge and implicit learning by both designers and machines.

References

- Alexander, C., (1964), *Notes on the synthesis of form*, Harvard University Press.
- Arlow, K., Neustadt, I., (2004), *Enterprise patterns and MDA: building better software with archetype patterns and UML*, Addison-Wesley.
- Beck, K., (2000), *Extreme programming explained: embrace change*, Addison-Wesley.
- Berry, D. C., Ed., (1997), *How implicit is implicit learning?*, Oxford University Press.
- Cockburn, A. (2002), *Agile Software Development*, Addison-Wesley.
- Brooks, F. P., (1995), *The mythical man month: essays on software engineering, 25th year anniversary edition*, Addison-Wesley.
- Champin, P.-A., (2001), “A model to represent design episodes for reuse assistance with interactive case-based reasoning”, in *Proceedings of the German Workshop on Case-Based Reasoning*, March 2001, Baden-Baden, Germany, pp189-197, Shaker Verlag, Germany.
- Do, E. Y., (2002), “Drawing marks, acts and reacts: towards a computational sketching interface for architectural design”, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 16, no. 3, pp. 149 -171.
- Dominowski, R. L., (1995), “Productive Problem Solving”, in *The creative cognition approach*, Smith, S. M., Ward, T. B., Finke, R. A., Eds., MIT Press.
- Egyed, A., Wile, S., (2006), “Support for managing design time decisions”, *IEEE Transactions on Software Engineering*, vol. 32, no. 5, pp. 299 – 314.
- Fowler, M. (1997), *Analysis patterns: reusable object models*, Addison-Wesley.
- Gelernter, D., (1998), *Machine beauty: elegance and the heart of technology*, Basic books, Perseus Books Group.

- Gigerenzer, G., Todd, P. M., (1999), *Simple heuristics that make us smart*, Oxford University Press.
- Glass, R. L., (2003), *Facts and fallacies of software engineering*, Addison-Wesley.
- Goel, A. K., (1997), “Design, analogy and creativity”, *IEEE Expert, Intelligent Systems and their Applications*, vol. 12, no. 3, pp. 62 – 70.
- Guindon, R., (1990a), “Designing the design process: exploiting opportunistic thoughts”, *Human Computer Interaction*, vol. 5, no. 2-3, pp. 305 – 344.
- Guindon, R., (1990b), “Knowledge exploited by experts during software-system design”, *International Journal of Man-Machine Studies*, vol. 33, no. 3, pp. 279 – 304.
- Guindon, R., (1992), “Requirements and design of design vision: an object-oriented graphical interface to an intelligent software design assistant”, in *Proceedings of the ACM Conference of Human Factors in Computing Systems (CHI '92)*, May 1992, Monteray, CA, USA, pp. 499 – 506, ACM Press.
- Harman, M., Swift, S., Mahdavi, K. (2005), “An empirical study of the robustness of two module clustering functions”, in *Proceedings of the Genetic and Evolutionary Computing Conference 2005 (GECCO '05)*, June 2005, Washington, DC, USA, pp. 1029 – 1036, ACM Press.
- Harmain, H. M., Gaizuaskas, R., (2003), “CM-Builder: a natural language based CASE tool for object-oriented analysis”, *Automated Software Engineering*, vol. 10, no. 2, pp. 157 – 181.
- Jackson, M., (1995), *Software requirements & specifications: a lexicon of practice, principles and prejudices*, ACM Press.
- Lawson, B., (2004), *What designers know*, Architectural Press.
- Liu, D., Subramaniam, K., Eberlein, A., Behrouz, H., (2004), “Natural language requirements analysis and class model generation using UCDA”, in *Proceedings of the 17th International Conference on Industrial and Engineering Application of Artificial Intelligence and Expert Systems (IEA/AIE 2004)*, May 2004, Ottawa, Ontario, Canada, Lecture Notes in Artificial Intelligence, pp. 295 -304, Springer-Verlag, Germany.
- Jung, C., (1926), *Psychological types*, (translated: Baynes, H. G.) Routledge and Kegan Paul.
- Larvet, P., Vallee, F., (2002), “UML developments: cost estimation from requirements”, in *Proceedings of the 7th International Conference on Software Quality*, June 2002, Helsinki, Finland, *Lecture Notes in Computer Science*, vol. 2348, Springer.
- Lo, S.-C., Chang, J-H., (2004), “Application of clustering techniques to software component architecture design”, *International Journal of Software Engineering and Knowledge Engineering*, vol. 14, no. 4, pp. 429 – 439.
- Machwe, A., Parmee, I. C. (2006), “Integrating aesthetic criteria with evolutionary processes in complex, free-form design”, in *Proceedings of the IEEE Congress on Evolutionary Computing (CEC '06)*, July 2006, Vancouver, Canada, IEEE Press.

- Mich, L., Garigliano, R., (2002), “NL-OOPS: a requirements analysis tool based on natural language processing”, in *Proceedings of the 3rd International Conference on Data Mining (Data Mining III)*, September 2002, Bologna, Italy, Wessex Institute of Technology Press.
- Mitchell, B., S., Mancoridis, S., (2006), “On the automatic modularisation of software systems using the Bunch tool”, *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193 -208.
- OMG, (2006), Object Management Group, available online: <http://www.uml.org/>
- Parmee, I. C., (2002), “Improving problem definition through interactive evolutionary computing”, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 16, no. 3, pp. 185 – 222.
- Parmee, I. C., (2005), “Human-centric intelligent systems for exploration and knowledge discovery”, *Analyst*, vol. 130, pp. 29 – 34.
- Parmee, I. C., Abraham, J. A., (2004), “Supporting implicit learning via the visualisation of COGA multi-objective data”, in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '04)*, June 2004, Portland, Oregon, USA, pp. 395 – 402, IEEE Press.
- Parmee, I. C., Cvetkovic, D., Watson, A. H., Bonham, C. R., (2000), “Multiobjective satisfaction within an interactive evolutionary design environment”, *Evolutionary Computing*, vol. 8, no. 2, pp. 197 – 222.
- Robillard, P. N., (1999), “The role of knowledge in software development”, *Communications of the ACM*, vol. 42, no. 1, pp. 87 – 92.
- Schwaber, K., Beedle, M., (2002), *Agile Software Development with Scrum*, Prentice Hall.
- Schooler, J. W., Melcher, J., (1995), “The ineffability of insight”, in *The creative cognition approach*, Smith, S. M., Ward, T. B., Finke, R. A., Eds., MIT Press.
- Seng, O., Stammel, J., Burkhart, D., (2006), “Search based determination of refactorings for improving the class structure of object-oriented systems”, in *Proceedings of the Genetic and Evolutionary Conference 2006 (GECCO 2006)*, July 2006, Seattle, WA, USA, pp. 1909 – 1916, ACM Press.
- Simon, H. A., (1973), “The structure of ill-structured problems”, *Artificial Intelligence*, vol. 4, pp. 181 - 201.
- Simon, H. A., (1996), *The sciences of the artificial*, MIT Press.
- Simons, C. L., Parmee, I. C., Coward, P. D., (2003), “35 years on: to what extent has software engineering achieved its goals?”, *IEE Proceedings – Software*, vol. 150, no. 6, pp. 337 – 350.
- Simons, C. L., Parmee, I. C., (2006a), “A comparison of genetic operators in conceptual software design”, in *Proceedings of the 7th International Conference in Adaptive Computing in Design and Manufacturing (ACDM '06)*, April 2006, Bristol, UK, pp. 67 – 72, Institute for People-Centred Computation, UK.
- Simons, C. L., Parmee, I. C., (2006b), “Multi-objective genetic algorithms in object-oriented conceptual software design”, in *Proceedings of the Artificial Neural Networks in*

- Engineering: Intelligent Systems Design (ANNIE '06)*, November 2006, St. Louis, Missouri, USA, American Association of Mechanical Engineers (ASME) Press.
- Smith, S. M., (1995), "Fixation, incubation, and insight in memory and creative thinking", in *The creative cognition approach*, Smith, S. M., Ward, T. B., Finke, R. A., Eds., MIT Press.
- Sperber, D., Wilson, D., (1995), *Relevance: communication and cognition*, 2nd Edition, Blackwell.
- Svetinovic, D., Berry, D. M., Godfrey, M., (2005), "Concept identification in object-oriented domain analysis: why some students just don't get it", in *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE '05)*, August 2005, Paris, France, pp. 189 - 198, IEEE Press.
- Turing, A., (1950), "Computing machinery and intelligence", *Mind*, vol. 59, pp. 433 – 460.
- Wang, L., Shen, W., Xie, H., Neelamkavil, J., Pardasani, A., (2002), "Collaborative conceptual design – state of the art and future trends", *Computer-Aided Design*, vol. 34, pp. 981 -986.
- Westcott, M. R., (1968), *Towards a contemporary psychology of intuition: a historical, theoretical and empirical enquiry*, Holt, Rinehart and Winston, Inc.
- Wirfs-Brock, R. J., McKean, A., (2003), *Object design: roles, responsibilities, and collaborations*, Addison-Wesley.
- Wirfs-Brock, R. J., (2006), "Looking for powerful abstractions", *IEEE Software*, vol. 23, no. 1, pp. 13 -15.
- Wu, J., Graham, T. C., (2005), "The software design board: a tool for supporting workstyle transitions in collaborative software design", in *Proceedings of the Engineering Human Computer Interaction and Interactive Systems: Joint Working Conferences (EHCI-DSVIS 2004)*, July 2004, Hamburg, Germany, pp. 363 – 382, Springer-Verlag, Germany.