1 INTRODUCTION

Design is a fundamental component of software development. Indeed, early lifecycle software engineering design is crucial as concepts and information discovered from the design problem domain are used as a basis for subsequent development activities. Because of this, inferior designs can lead to deleterious and costly down-stream consequences. Therefore any improvement in the traceability, structural integrity and elegance of software design has significant potential for enhancing software development productivity. However, there is a body of evidence to suggest that early lifecycle software design is a demanding and non-trivial task for software engineers to perform, while current computational tool support for early lifecycle software design is limited. Thus to address this computational tool support limitation, this thesis investigates the potential of interactive evolutionary search and complimentary computational intelligence to enable the exploration and discovery of software designs that are both useful and interesting to the designer and relevant to the design problem domain.

1.1 Early Lifecycle Software Design

Early lifecycle software design is an intensely people-oriented activity wherein concepts and information relating to a relevant design problem domain are identified and evaluated. For many years, such software design concepts and information have been widely modelled using the object-oriented paradigm by means of the 'class' construct. The Unified Modelling Language (Object Management Group, 2010a) is the standard modelling language of the object-oriented paradigm and is extensively used by software designers to visualise and record classes as well as other aspects of software design.

However, there is much evidence to suggest that the act of early lifecycle software engineering design is non-trivial and demanding to perform. Brooks (1995) believes *"the hard part of building software to be the specification, design and testing of this software construct, not the labour of representing it and the testing of the fidelity of the representation"*. Guindon (1990) reports software design as an ill-structured problem in which the software design process is essentially opportunistic. As a result of empirical observations of many software designers, Guindon suggests that activity of software design is rarely sequential in reality, frequently iterating and relying heavily on opportunistic thoughts of the designer. Zannier *et al.* (2007) have observed both

rational and opportunistic / naturalistic design decision making tactics in software design. Based on empirical studies spanning more than twenty years and more than twenty industrial companies, Petre (2009) suggests that software design problems *"are often 'wicked': too big, too ill-defined, too complex for easy comprehension and solution. Sometimes the problems are only fully understood after they are solved. Solving such problems is rarely a matter of 'brute force' or routine"*. Because of this, Glass (2003) goes further to suggest that the complexities of some software designs may be beyond human comprehension. In an attempt to explain this, Glass asserts that there are usually many potential solutions to a design problem, although there is seldom one best design solution. Glass echoes Guindon's observation that software design is a complex iterative process, and suggests that initial design solutions will likely be wrong and certainly not optimal.

There is also evidence to suggest that individual software designers may be blessed with varying degrees of design talent. Even for experienced designers, Glass (2003) notes that designer performance may vary from 28:1 from the best engineers to the worst. Petre (2009) cites Boehm's (1981) suggestion that individual developers differ in productivity by 10 to 30 times to advocate that, as a result, software design expertise is the crucial commodity in software development today. Curtis (1998) also notes a range in design talent, and observes that only "super-designers" can reason across the full breadth and depth of complex, ill-structured design problems in order to fully consider the consequences and implications of design decisions. From the field of education, there is evidence to suggest that software design is difficult to learn. In teaching object-oriented early lifecycle software design to over 740 university undergraduate students over 135 software design problems, Svetinovic *et al.* (2005) observe that with respect to concept identification, *"some students just don't get it"*. Blaha *et al.* (2005) report a software design study of 21 academic institutions across four countries and observe that many students have difficulty resolving ambiguities within ill-structured design problems. However, as the students gain design experience over time, they are increasingly able to deal with design ambiguity.

To help overcome these difficulties, mature heuristics and manual strategies for the discovery of potential design solution concepts from the problem domain are available. For example, Wirfs-Brock and McKean (2003) advocate a responsibility-driven design approach, in which the responsibilities of a design concept are captured in

the context of collaborations with other concepts. Such concepts map directly onto Unified Modelling Language (UML) classes, and are recorded via paper 'CRC cards' (where the first 'C' denotes concept or class name, 'R' denotes the concept's responsibilities and the second 'C' denotes the collaborations). Wirfs-Brock and McKean advise software designers to identify candidate concepts *"somewhat systematically"* and offer manual search strategies that *"make educated guesses about the kinds of inventions that you will need based on the nature of your application"*. In a different approach, early lifecycle software design patterns (e.g. Fowler, 1998, Arlow and Neustadt, 2004) have been proposed, which have led to domain-driven design approaches (e.g. Evans, 2004). Software design patterns and domain-specific approaches have made a significant contribution to help mitigate the difficulties of early lifecycle software design (particularly with respect to design reusability and maintenance), but require design expertise to successfully match to a specific design context. Furthermore, being well-established and historical, it would appear that patterns and domain-driven design do little to enhance innovation and creativity in software design.

In addition, object-oriented software development methodologies offer software designers a variety of workflows by which they might conduct their early lifecycle software design activities, the artefacts they might produce and the development roles required. In a comprehensive survey review of object-oriented development methodologies, Ramsin and Paige (2008) suggest three categories of methodology: seminal, integrated and agile. Seminal methodologies (e.g. Booch, 1994) are early incarnations, while integrated methodologies (e.g. Jacobson *et al.*, 1999) combine and integrate many early ideas to suggest a more comprehensive, standardised approach. Agile methodologies (e.g. Beck, 2000) differ from the previous two categories by placing emphasis on people and their interactions, customer collaboration and responding to change. While methodologies in all three categories describe iterative and evolutionary development to some extent, it is agile methodologies that place software evolution in the face of inevitable change at the heart of development activities (Boehm and Beck, 2010). Indeed, such is the interest in software evolution that the Institute of Electric and Electronic Engineers (IEEE) have initiated a number of International Workshops on the Principles of Software Evolution (e.g. IEEE, 2005). However, with respect to guidance to software designers to overcome the difficulties of early lifecycle

software design, it would appear that development methodologies offer little support. While object-oriented development methodologies specify in detail the development workflows and artefacts produced, they pay less attention to guidance for designers, for example, in terms of fitness evaluation of artefacts. In other words, while methodologies thoroughly address the 'who', 'what' and 'when' of software development, it would appear that the 'how' is less well addressed.

Early lifecycle software design is significant because although there may be some debate concerning the sequential versus iterative and evolutionary nature of the software development lifecycle (Boehm and Beck, 2010), consequences of early lifecycle design decisions persist and may propagate as development progresses. It seems likely that poor requirements capture and software design can have significant deleterious downstream consequences for software development (Damien *et al.*, 2005). The impact of early lifecycle software designs of poor structural integrity and poor traceability to the problem domain appears to be far reaching and detrimental to subsequent development activity, because the economic cost of rectifying errors in software can rise exponentially as development proceeds downstream over time (Boehm, 1981). Thus, given the difficulties of early lifecycle software design, the potential of computational tool support to assist the software designer is considerable. It seems likely that even modest gains in the traceability and structural integrity of early lifecycle software designs might be amplified as the development lifecycle progresses to yield significant productivity gains. However, the extent of current computational tool support for the early lifecycle software designer is mixed. Some UML-based modelling tools (e.g. Object Management Group, 2010b) seem to do little to proactively support the early design process. Rather, they appear to provide an opportunity for the designer to formally record the output of their design decisions after the cognitive hard work of following manual heuristics and tactics has been done. Other commercially available UML modelling tools e.g. IBM Rational Software Architect (IBM, 2010), Enterprise Architect (Sparx Systems, 2010) do provide considerable utility to the designer through, for example, design visualisation, dependency analysis, verification of UML version 2.n well-formedness rules, automated code generation, reverse engineering, version control and refactoring suggestions. Nevertheless, such UML modelling tools rarely suggest candidate solution classes derived from the problem domain to the designer, and then follow up by evaluating them. Indeed, in the terms of Brooks (1995), this would appear

to be tool support for the labour of representing software designs and the testing of the fidelity of the representation, rather than the specification and design of the software, which Brooks suggests *"is the hard part"*.

1.2 Computational Intelligence

Given that manual iterative and evolutionary approaches to software design and development are now predominant (IEEE, 2005; Boehm and Beck, 2010), could computational intelligence (such as evolutionary computing and machine learning) offer potential as the basis of computational tool support for a more automated early lifecycle software design process?

Evolutionary computing draws inspiration from natural evolution. In nature, populations of individual organisms exist within an environment which is typically subject to constant change. To respond well to this change, a population of organisms must adapt as generations live and die. This is achieved by a combination of (1) environmental pressures which tend to select only the fittest to reproduce into the next generation, and (2) mechanisms for sexual reproduction which generate a degree of diversity in the offspring. In computer science, evolutionary computing draws upon natural evolution and uses iterative processes to simulate evolutionary among a~~, such as the growth or development of a~~ population of individuals. As individuals of superior fitness are selected, they breed offspring before dying and so after numerous generations, the population as a whole is then guided in a stochastic search to achieve a desired end. ~~According to Eiben and Smith (2003), t~~The notion of 'genetical or evolutionary search' was proposed by Turing in ~~as early as~~ 1948. Proposals for evolutionary programming (e.g. Fogel *et al.*, 1966) were reported in the USA while in Germany, evolutionary strategies (e.g. Rechenberg, 1973) were suggested. Again in the USA, a proposal for genetic algorithms by Holland (1975) was later popularised by Goldberg in his 1989 book. After some years of separate development, the 1990s saw much cross-fertilisation of research ideas and the above different streams came to be known as 'dialects' of evolutionary computing. Around that time, the ideas of genetic programming also arose, as championed by Koza (1992). Since then, the field of evolutionary computing has attracted wide research attention resulting in its application to a plethora of single-objective and multi-objective automated search problems. Useful surveys of the applications of multi-objective evolutionary algorithms have been

conducted by Coello Coello (2000), Deb (2001) and Coello Coello *et al.* (2007). ~~In evolutionary computing generally, Goldberg (1989) highlights the need to balance exploration of the search (through the preservation of diversity) versus exploitation (through selection of the fittest solutions). Indeed, Goldberg points out that achievement of such balance gives genetic algorithms robustness in the face a wide array of search problem domains.~~

Evolutionary computing has shown much potential in a variety of fields of design. For example, Bentley (1999) surveys the use of evolutionary design by computers in the fields of engineering design, computer art and artificial life. Parmee (2001a) reports the successful interactive evolutionary search and exploration in a range of engineering design projects. Jain *et al.* (2007) report the use of evolutionary computing across a wide range of complex systems design, while a more recent survey of the utility of evolutionary computing for the design of biological artefacts, art, computational embryogeny and engineering is provided by Hingston *et al.* (2008). Indeed, the potential of evolutionary computing in conceptual design has been researched by Parmee *et al.* (2007) who surveyed the diversity of early lifecycle (conceptual) design across a wide range of design fields with respect to computational design environments that engender design discovery. Nevertheless, despite the potential and wide application of evolutionary computing in design generally, for any computational design support tool to be effective, it must support and enable the designer and the design process. Consistent with previous research by Parmee (2001a), it is conjectured that it is currently impossible to exclude the designer to fully automate early lifecycle software design, given the richness and complexities of design. Rather, it is necessary to include the designer during design and for this to be effective, the human designer and the computational support tool must collaborate interactively (Parmee *et al.*, 2002a). In reality, effective two-way interaction constitutes an open system, in the sense of a collection of collaborating software, hardware and human components, working together cooperatively towards a common goal. This incorporation of people-centred interaction in evolutionary algorithms has been described by Tagaki (2001) as Interactive Evolutionary Computation (IEC). Tagaki explains IEC as a fusion of evolutionary computation and human fitness evaluation and reports the application of IEC to a wide variety of areas, e.g. arts and animation, music, virtual reality, image processing, data mining, robotics and various fields of design.

Despite considerable advances over a wide range of design fields, challenges posed by the effective use of evolutionary computation as the basis of computationally intelligent support for design remain. For example, whenever open, interactive, user-centred support for design is required, autonomous software agents and machine learning techniques have shown considerable potential to stimulate designer engagement and so help reduce interaction fatigue. However, research into the use of proactive user-interface software agents to shield the user, filter repetitive and fatiguing tasks and so promote an engaging interactive experience, is on-going (Wooldridge, 2009). In addition, research has also been conducted into autonomous agents that employ machine learning techniques such as clustering and case-based reasoning to learn, for example, qualitative design aspects such as design aesthetics (Machwe and Parmee, 2009), or fuzzy-rule extraction of qualitative design evaluations (Brintrup, 2008). It is also interesting to also note that although reports of machine learning techniques across the software engineering lifecycle have been recently emerging (e.g. Zhang and Tsai, 2005, Zhang, 2010), reports specific to early lifecycle software design are less readily available.

Given that the utility of these approaches is proving beneficial across several different fields of early lifecycle design, could interactive evolutionary computing be a basis for computational tool support in software design? Certainly, interactive evolutionary search, when combined with other machine learning techniques, affords opportunities for not only synthesizing a space of early lifecycle UML class design solutions, but also their evaluation by a combination of quantitative objective fitness functions and qualitative subjective evaluation. As an open system, opportunities for both human and machine to learn and jointly steer the direction of search and exploration present themselves. Furthermore, a computationally intelligent search-based approach is consistent with the manual search strategies of Wirfs-Brock and McKean (2003) for the discovery of potential UML class design solutions.

1.3 Research Contributions

The thesis is based upon the hypothesis that computationally intelligent tool support can greatly assist the designer with the complexities of identifying and specifying concepts and information from the design problem domain in early lifecycle software design. Specifically, it is hypothesised that interactive evolutionary computing affords

significant opportunities for combined quantitative and qualitative multi-objective search and exploration of the early lifecycle software design solution space. It is also hypothesised that task-focussed (single function) and autonomous software agents offer great potential for the facilitation of not only collaborative two-way interaction, but also machine learning techniques which may reveal new knowledge relating to the nature of software design elegance, whilst enhancing designer interaction by reducing user fatigue. Finally it is expected that such computational tool support for the software designer may afford opportunities for the discovery of useful, innovative and elegant early lifecycle UML class designs, thus helping to produce gains in the traceability and structural integrity of such designs, which, in turn, may yield significant development productivity gains.

The presented work focuses on the development of an early lifecycle software design interactive evolutionary environment, and investigations into its subsequent evaluation. It covers the following research areas:

- How to represent the design problem and early lifecycle design solution such that they are sufficiently abstract for human comprehension but also effective for computational evolutionary search.

  A novel object-based representation of the design solution is presented in chapter 4, and is directly traceable to the design problem. The novel object-based representation underpins all subsequent investigations.
- How to effectively explore and exploit the software design solution search space using evolutionary computing to produce useful and innovative UML class software designs.

  Novel genetic operators underpinning single and multi-objective evolutionary search are introduced in chapter 6, and developed significantly in chapter 9 to incorporate qualitative elegance intentions.
- Within early lifecycle software design, how to facilitate collaborative designer / computer interaction using autonomous software agents.

  The use of global and local search agents, and a design episode controller agent that together promote effective interaction, are presented in chapter 7 and investigated empirically in chapter 8. An elegance agent, that captures and learns designer elegance intentions, is introduced in chapter 9.

- What measures can be formulated to capture software design elegance intentions.

  Four novel measures of software design elegance based on design symmetry and distribution are presented and investigated in chapter 9.
- How machine-based quantitative fitness metrics can be effectively integrated with qualitative designer evaluations in multi-objective evolutionary search.

  A dynamic multi-objective search mechanism is introduced and investigated in chapter 9.
- The manner in which interaction fatigue can be reduced by machine learning techniques such as dynamic interaction and reward-based learning.

  A dynamic, adaptive interactive interval approach is introduced in chapter 9. Also, a reward-based learning mechanism is introduced in chapter 9 which both reflects designer elegance intentions and drives dynamic multi-objective search. Both research aspects are investigated and reported using an autonomous elegance agent in chapter 9.