The Truth is Out There: Reflections on Search in Software Engineering

Christopher L. Simons

Department of Computer Science and Creative Technologies, University of the West of England, Bristol, BS16 1QY, United Kingdom

Abstract

In the popular science fiction horror drama TV series "The X-Files", two FBI agents (Mulder and Skully) investigate unsolved case files relating to emerging paranormal phenomena and possible alien life. Many explanations and conspiracy theories abound. Although the intrepid investigators struggle to put the disparate pieces together, they believe that "the truth is out there".

Search-based software engineering has attracted much research attention recently and many theories also abound relating to the application of metaheuristic search techniques to software engineering problems. Some 15 years since the term 'search-based software engineering' was suggested, it is perhaps timely to reflect on some of these emerging phenomena in the field of search-based software engineering and examine some of the theories, fallacies and facts in a wider software engineering context. Is there truth out there?

This presentation suggests some possible fallacies of search with respect to software engineering, before reviewing some more established facts about the progress of search-based software engineering, 15 years on. The application of search-based software engineering techniques within different phases of the software engineering life cycle is discussed, with a particular emphasis on agile development methodologies. Finally, attempts are made to put the disparate pieces together to speculate on areas of future industrial adoption of search-based software engineering.

1. Introduction

In the popular science fiction horror drama TV series "The X-Files", two FBI agents (Mulder and Skully) investigate unsolved case files relating to emerging paranormal phenomena and possible alien life. Many explanations and conspiracy theories abound in an attempt to explain these phenomena. Although the intrepid investigators struggle to put the disparate pieces together, they resolutely believe that "the truth is out there".

Email address: chris.simons@uwe.ac.uk (Christopher L. Simons)

Search-Based Software Engineering (SBSE) has attracted much research attention recently and many theories also abound relating to the application of metaheuristic search techniques to software engineering problems. Some 15 years since the term search-based software engineering was suggested [1], it is seems appropriate to reflect on some of the emerging phenomena in the field of search-based software engineering, and examine some of the theories, fallacies and facts in a wider software engineering context. Is the truth out there?

2. Some Ideas Out There - Possible Fallacies?

While many ideas have been suggested to explain phenomena in various aspects of software engineering, there are some that deserve close scrutiny because of their particular resonance to the application of metaheuristic search. For example, Glass [2] examines the idea that *"you can't manage what you can't measure"*. A derivation of *"you can't control what you can't measure"* originally proposed in 1986 by De Marco [3], this idea is based on the reasonable premise that managing software development in the presence of data is generally more effective than managing in its absence. However, some aspects of software engineering are more effectively managed qualitatively rather than quantitatively, and De Marco has more recently revised his thoughts [4] to focus on the delivery of software based on business 'value' - something difficult to measure and highly qualitative. Thus Glass asserts the original idea is a fallacy [2]. If so, this fallacy resonates on the performance of metaheuristic search which is attempting to optimize solutions based on (mostly) quantitative measurements as objective fitness functions.

Phenomena have been observed in agile software development methodologies. A full discussion of agile development methodologies is beyond the scope of this paper, yet such is the significance of their contribution to the field and predominance in contemporary software development practice, the existence some agile ideas put forward also merit scrutiny. For example, in a rigorous and colourful critical appraisal, Meyer [5] suggests there are 'bad and ugly' ideas in agile approaches. He cites the deprecation of upfront tasks in agile such as requirements engineering, architecture and design, and feature based development among others as ideas whose truth is problematic in reality. Moreover, the question *"is design dead?"* within an agile context has also been discussed [6], and not fully resolved. It seems likely that the full truth in areas such as upfront requirements, architecture and design is more subtle, and Meyer speculates on the influence of increasing software system scale as a causative factor in these areas. However, the resonance of these 'bad and ugly' agile ideas within search perhaps seems to be quite appropriate, as much research attention has been paid to these aspects of software development (e.g. see [7], [8] and [9] respectively).

A number of phenomena relating to search within software engineering have also been observed. For example, perhaps reflecting that search originally emerged mainly as an optimization technique, quantitative software metrics have been reported as good candidates for objective fitness functions [10]. In constrained optimization problem domains, e.g. the search-based optimization of test case suites for branch coverage, the fitness measure is typically immediately apparent. However, a number of software engineering texts have emerged discussing the limitations of metrics generally in software engineering. For example, Fenton and Bieman [11] raise questions concerning what metrics actually represent with regard to what exactly is being measured, and *"how to make measurable what is not measurable"*. Moreover, Cinneide et al. [12] recently report that considerable disagreement is found in search trajectories using differing cohesion metrics that claim to measure the same concept (i.e. class cohesion), when used in search over the same problem domain. The results obtained cast doubt on the ability of various cohesion metrics to act as universal objective fitness functions in search. In a separate empirical study, Simons et al. [13] captured software engineers' qualitative evaluation of various design qualities over a range of software designs, and compared them with quantitative metric values where such metrics aim to reflect similar qualities. Little or no correlation between the two was found. It seems possible that the idea of software metrics as fitness functions in search could be fallacious under certain circumstances - for example, when many concerns are being measured simultaneously, or when qualitative evaluation is a concern.

While it is widely understood that representation should be appropriate for the problem domain (e.g. see [14]), the notion that the representation is comprehensive is likely to be a fallacy. It is more typical for a solution representation to enable search components (such as fitness evaluation and diversity operators) for reasonable execution performance rather than model all aspects of the problem being investigated. As Meigan et al. [15] point out, there are limitations with the representation model in that a solution individual may be *"an approximation of complex problem's aspects"* and a *"simplification for model tractability"*.

Although not fallacies, other ideas have emerged whose complete semantic may have been somewhat truncated over the course of time. For example, in support of decision making in software engineering, many multi-objective evolutionary algorithms have been applied (e.g. see [16], [17]). While such algorithms undoubtedly make a significant contribution, Deb [18] points out that from a practical standpoint, there are two steps involved in an ideal multi-objective optimization procedure:

1. *"Find multiple trade-off optimal solutions with a wide range of values for objectives"*, and
2. *"Choose one of the obtained solutions using higher-level information, requiring various subjective and problem-dependent considerations"*.

While considerable research attention has been directed to the first step, it seems likely that less has been aimed at the second. In a population-based multi-objective search, it is typical to employ population sizes of hundreds of solution individuals. Sources investigating the choice of one of the obtained solutions from a search population based on higher-level information are less readily available in the research literature.

3. Some Facts, or What We Do Know

A large number of ideas capturing what we do know about software engineering have been put forward. However, it is interesting to focus on some that at first glance might not seem obviously pertinent to search, but upon reflection, acquire relevance. For example, according to Glass [2], the following two facts about software engineering relate to software engineers:

- *"The most important factor in software work is not the tools and techniques used by the programmers, but rather the quality of the programmers themselves"*, and
- *"The best programmers are up to 28 times better than the worst programmers."*

Interestingly, Glass quotes some software engineering folklore to support these facts (e.g. Sackman, 1968, [19], and Boehm, 1975, [20]). In addition however, more recent research in 2014 into measuring development and programming skills [21], also reports a significant range of skills to be found in practicing software engineers. In agile methodologies, the importance of software engineers in software development is also emphasised. As the agile manifesto [22] states, its authors have come to value *"individuals and interactions over processes and tools"*. Given this understanding of the fact, it is interesting to speculate on the role of the software engineer in SBSE [23].

Given this lack of emphasis on processes and tools, where might this leave ideas about the development tools use by software engineers? At first glance, perhaps surprisingly, Glass [2] presents as fact:

- *"Software developers talk a lot about tools, but seldom use them".*

In discussing this idea, Glass goes further to assert that developers talk a lot about tools, evaluate some, but use few. Glass speculates that this might be due, at least in part, to an occurrence of the "not-invented-here" phenonemon, although pressing delivery timescales may also have a role to play in not allowing software engineers the time to thoroughly research and evaluate new tools that might enhance productivity. Particularly with semi-autonomous tools such as decision-support or recommender systems, lack of engineer trust can be a barrier to tool adoption, as pointed out in [15], for example. Regarding trust in software tools, valuable lessons can be drawn from the field of multi-agent systems. For instance, Klien at al. [24] report a number of challenges for making automation a "team player" in any joint human-agent activity. Clearly, the software engineer and the tool must use common ground to work jointly to agreed goals, but other factors are important too. For example, *"To be a team player, an intelligent agent – like a human – must be reasonably predictable and reasonably able to predict others actions"*, while at the same time *"agents must be able to observe and interpret pertinent signals of status and intentions."* Given the significant range in software engineer competencies discussed above, successfully achieving trust in search-based software tool support is clearly non-trivial.

Given the above phenomena of software engineering, it is perhaps not surprising that there appears to be a gap between SBSE tool innovation and practitioner adoption. In a recent article [25], Gregory et al. report the results of an empirical survey into the challenges that practicing software engineers face in an agile context, and consistent with the human-centred 'facts' presented on software engineers above, discover that organizational, cultural and team-based challenges predominate, although the challenges of scale and complexity also emerge. Interestingly, Freudenburg and Sharp suggest a 'top ten' of burning research questions from practitioners [26]. Little consistency is found between this 'top ten' and the wider SBSE research literature. Top research questions from practitioners relate (in order of numbers of publications) to large scale projects, self-organisation and team co-location.

To go some way to explain this gap, it seems possible that there could be a historical lack of emphasis on empirical research in SBSE. The SBSE literature has not always reported empirical studies in abundance, although it is interesting to note that the past five years have seen an small upsurge in some interesting empirically-based research publications. A few notable examples include Hemmati et al. [27] who conducted an industrial case study into enhanced test case selection approach for model-based testing. Amannejad et al. [28] conducted an industrial case study to investigate what parts of a software system should be tested in an automated fashion and what parts should remain manual, and suggest that focus should be on return on investment (ROI). Wang et al. [29] also used an industrial case study, but used industrial data to formulate a novel fitness measure for test case prioritisation in a multi-objective approach to product line testing. Simons et al. [30] employed the services of software designers to interact with an Ant Colony Optimization search of software designs using both quantitative metrics for design coupling and the designers' qualitative evaluation of design elegance. Marculescu et al. [31] conducted an initial industrial evaluation of interactive search-based testing for embedded software using problem-domain experts who had little or no knowledge of SBSE. Recently, Araaujo et al. [32] propose an architecture based on interactive optimization and machine learning for application to the next release problem, derived from not only implicit, tacit preferences of the decision maker, but also quantitative measures of release dependencies.

It is also interesting to focus on some facts concerning what we know about search that might have resonance to software engineering. Firstly, it is well known that both effectiveness and efficiency are important in the performance of search. However, while fitness values and/or attainment surfaces for Pareto-fronts are typically reported, execution time has not always been reported in results of many papers in the SBSE literature. Algorithm performance with respect to parameter tuning has been investigated (e.g. [33], [34], [35]) as has parallel execution (e.g. [36]). Langdon and Harman report a considerable execution performance gain in one case study using genetic programming [37]. Nevertheless, performance times may not always be a significant part of the contribution of some papers, although if an empirical study or case study is described, performance times can be crucial.

It is also evident that the provision of additional benchmark problem instances would assist more rigorous comparison of various metaheuristic search approaches. Some issues relating to software engineering repositories are discussed by Rodriguez et al. [38].

Given that an important factor is the quality of software engineering is human - the developers and programmers - the role of human interactivity in search has been discussed previously (e.g. see [23]). Interactive search extends the role of search from solely optimization to the generation of insight and knowledge discovery within the problem instance being searched [39]. In such interactive search, it is typical to combine quantitative fitness determined by software metrics with qualitative developer evaluation (e.g. [9]) to obtain a rich assessment of solution fitness.

4. Search in (Agile) Practice

In this section, we look at various practices within the software engineering, and examine the phenomena associated with current search application. We also speculate on ways in which search might show potential. However, rather than taking a classical waterfall view of the life cycle (e.g. requirements, design, code, test), agile practices are examined because of their wide adoption and acceptance in the software engineering community (e.g. see [40], [41]). Considering what constitutes a practice in software development, Meyer [5] suggests: *"A practice has to be an activity or mode of working, but with a special twist: repeated application. In the absence of repetition, we may have an interesting technique, but it is not a practice unless it is performed regularly (in the case of an activity) or enforced systematically (in the case of a mode of working)"*.

Drawing upon this assertion and other sources on the agile community [42], [43], [44], the following subsections examine engineering and development practices (in no particular order) with respect to ways in which search has been applied. Also, where appropriate, we speculate on novel ways in which search might be applied. Agile practices examined include iteration planning, the retrospective, collective code ownership, continuous integration, pair programming, refactoring and test-first programming.

(Some development practices such as modes of working have not considered in this paper because they focus exclusively on engineer behavior and organisation, e.g. physical co-location of the engineering team, ready access to client stakeholders, informative workspaces etc. For such modes of working, many communication focused tools are available to assist with the manual organisation of planning, tracking progress and release of software; e.g. see [45]).

4.1. Iteration (Sprint) Planning

The planning of a delivery iteration, or 'Sprint' [42], involves examining the product backlog to produce a backlog of tasks appropriate for delivery at the end of an iteration. The product backlog is a prioritized queue of features and technical capability required of the software product-to-be, whereas the sprint backlog is a subset of the product backlog chosen for deployment and delivery at the end of the sprint.

As part of iteration planning, estimating task duration can be addressed by the 'Planning Game' [43], or 'Planning Poker' [42]. The Planning Game is a cooperative game wherein business stakeholders define task priority and developers estimate task duration. 'User stories' are typically used to describe feature tasks; metrics such as the 'story point' [46] have been proposed to measure story size. Planning Poker involves participants picking duration estimates from preset values, with each participant revealing their estimate in turn.

In the SBSE arena, iteration planning has been addressed in terms of 'Software Release Planning', and has attracted much research attention recently. For example, Aguila et al. [47] integrate multi-objective search components into a requirements management tool as part of a three stage process to assist the human planner. Del Sagrado et al. [48] report promising results for ant colony approach to optimize requirements selection. Pitangueira et al. [49] incorporate an assessment of cost, value and risk to reduce stakeholder dissatisfaction in multi-objective requirement selection. Araújo et al. [32] propose an architecture for interactive optimisation combined with machine learning for requirements planning and report promising outcomes in an empirical investigation. A useful survey of the area published in 2015 can be found at [7].

The attention devoted to iteration planning using search seems relevant and useful to this agile practice, and recent work using search-based decision support is encouraging. However, research appears to be inspired principally by quantitative optimization rather than by a game-based approach, or for instance, 'Planning Poker'. It is interesting to speculate on ways in which interactive metahueristic search could assist with game-based approaches, e.g. poker, particularly when combined with other forms of machine learning.

4.2. Retrospective

A sprint retrospective is a practice to review successful (and less successful) development aspects at the completion of the sprint. According to Meyer [5], *"the purpose is what we find at level 5, 'optimizing' of the CMMI: integrating into the process a feedback loop so that it can improve itself"*. The CMMI, or Capability Maturity Model Integration, describes five levels of process maturity from level 1, 'initial', to level 5, 'optimizing' [50].

In the SBSE arena, references to search-based optimization of sprint activities in a retrospective are not readily available in the literature. At first glance, this is perhaps surprising given that a primary focus of search is optimization. In an attempt to explain this, we firstly note that given the people-centred, documentation lightweight nature of agile processes, meta-modelling of agile processes is not immediately apparent in the literature. In other development methodologies, however, models of the development process have been proposed and widely adopted e.g. CMMI [50] and the International Standards Organisation process assessment approach Software Process Improvement Capability Determination (SPICE) [51]. In the arena of service-oriented architectures (SOA), a useful survey of service-based development process models is offered by Lane and Richardson [52].

It is fascinating to speculate on ways in which search might be applied to optimising agile processes as part of the sprint retrospective. Solution representation as a model of the development process presents challenges but building on CMMI and ISO 15504 might offer a starting point. Formulating quantitative and qualitative fitness measures relating to sprint velocity and achievement seem conceivable. Perhaps an interactive search approach could provide feedback, insight and discovery for adaptive agile development process improvement?

4.3. Collective Code Ownership

Although the phenomenon of collective code ownership has existed for many years, a number of authors advocate the practice of shared code ownership within an agile context (e.g. [43], [44]). These advocates point to shared code ownership as assisting other practices such as continuous integration and pair programming. However, others [53] point out that other models can exist e.g. single person ownership of designated source code. Whatever model is followed, the role of coding standards is crucial in this practice [54].

In the SBSE arena, references to collective code ownership are not immediately apparent. It seems possible that this agile phenomenon is closer to a mode of working rather than a practice, although the role of coding standards has implications for refactoring - this is discussed below.

4.4. Continuous Integration

Integrating the various components of a software system involves compilation, deployment and running of tests against the various changes to the code base made by team members. The larger the number of code changes, the greater potential difficulty of the compilation, build and deployment. Proponents of this agile practice (e.g. [43]) point out that regular and continuous integration builds can be difficult to achieve, but are highly beneficial in providing instant feedback on the status of system compilation and build.

In the SBSE arena, references to continuous integration are also not immediately apparent, although Harman ventures the opinion that the deployment process might be dynamically optimised [39]. However, it is interesting to further speculate if search-based approaches could assist in resolving difficult cases of build integration based on the dependencies resulting from numerous changes.

4.5. Pair Programming

Pair programming involves placing *"...two people sitting at one machine"* while programming [5]. While Beck [43] suggests that all production programs are developed in this manner, others [44], [42] suggest that pair programming is best practiced in periods of intense communication between programmers, perhaps when debugging an elusive error, interpreting a complex requirement or evaluating a design trade-off judgement.

In the SBSE arena, references to pair programming are also not immediately apparent. However, it is interesting to observe that attempts have been made use coevolution to assist in the automation of software correction (by coeolving tests with program code) [55]. It is interesting to speculate if a coevolutionary approach could be harnessed interactively for the practice of pair programming.

It is also most interesting to speculate on the ways in which genetic improvement (e.g. [56]) might be applied interactively for pair programming. In an interesting rethink of genetic improvement programming, White and Singer [57] argue for new ways forward for genetic improvement programming including *"program transformation, translation, cloning; code scavenging and recombination"*, which seem attractive possibilities for interactive application to pair programming.

4.6. Refactoring

The phenomenon of refactoring was first proposed some 25 years ago [58] and has been subsequently adopted as an agile practice. According to Fowler [59], *"Refactoring is the process of changing a software system in such a way that is does not alter the external behaviour of the code, yet improves its internal structure"*. Recent work has questioned that assertion however, suggesting that minor changes in semantic code behaviour may be tolerated for the sake of improved software quality. Moreover, Hafiz and Overby [60] suggest that it is a myth that manual refactoring tools are robust; rather they can unintentionally introduce semantic code changes and errors in the pursuit of improved code quality.

In the SBSE arena, refactoring has attracted significant attention. For example, Mkaouer et al. exploit high dimensional search to find trade-offs among up to 15 objectives in (semi-)automatic refactoring [17]. Amal et al. use machine learning and search-based software engineering to cope with ill-defined fitness functions [61]. In a further example, Simons and Smith use both pheromone and anti-pheromone in an ant colony optimization approach to software design refactoring [62]. In this approach, antipheromone is used to steer search away from poor solutions, which pheromone steers search towards superior solutions.

Although examples are not abundant in the literature, it is interesting to speculate how coding standards and collective code ownership might be exploited in search-based refactoring. One interesting proposal by Langdon [63] suggests applying an interactive genetic alogrithm to find good C source code layouts using GNU Indent. Given that coding standards and shared code ownership can facilitate continuous integration and refactoring, contributions of SBSE in this area would be most welcome.

4.7. Test-First Programming

In this practice, Beck [43] advises us to *"write a failing test before changing any code"*, thus emphasising the importance of evaluation in the development of software. After the test fails, development consists of fixing the failed test and refactoring where necessary. However, test-first programming has been criticised by Meyer [5] who points out that a test is one (important) part of a requirement; other requirements (perhaps expressed as 'user stories') have an important contribution to development too.

In the SBSE arena, Harman points out that generating test data is a demanding, and that *"... techniques to automate test data generation must cater for a bewildering variety of functional and non-functional test adequacy criteria and must either implicitly or explicitly solve problems involving state propagation and constraint satisfaction"* [64]. Harman then goes on to explain how SBSE might assist in this situation. However, more recently, others have questioned the ways in which automatic test generation helps programmers. For example, to investigate this issue, Fraser et al. [65] performed two controlled experiments comparing 97 subjects split between writing tests manually and writing tests with the aid of an automated unit test generation tool. While tool support led to a improvement in aspects such a code coverage, Fraser et al. report no measurable improvement in the number of bugs found be developers. Interestingly, it appears that for the engineers writing tests with the aid of the generator, there is a learning curve to be mounted in understanding the tests generated, and this might be followed by possibly discarding some poor quality generate tests. Also, high branch coverage may not necessarily lead to increased fault detection. Fraser et al. also suggest that the interaction during manual and automatic test generation is key to effectiveness - which is consistent with many of the interactive phenomena observed above in the previous section.

5. Concluding Reflections

What might our fictional FBI agents (Mulder and Skully) make of the ideas and theories abounding in search-based software engineering? It seems likely that our investigators might be very impressed with the wide range of search ideas and algorithms proposed for application in software engineering. But perhaps they might also be a little surprised at the gap between such innovative ideas and tool adoption in practice. It's also possible that they might also hope that as a sign of a maturing field, this gap may begin to close.

References

[1] M. Harman, B. F. Jones, Search-based software engineering, Information and software Technology 43 (14) (2001) 833–839.

[2] R. L. Glass, Facts and fallacies of software engineering, Addison-Wesley Professional, 2002.

[3] T. DeMarco, Controlling software projects: Management, measurement, and estimates, Prentice Hall PTR, 1986.

[4] T. DeMarco, Software engineering: an idea whose time has come and gone?, IEEE Software 26 (4) (2009) 96.

[5] B. Meyer, Agile! the good, the hype and the ugly, Springer Science & Business Media, 2014.

[6] M. Fowler, Is design dead?, Software Development (San Francisco 9 (4) (2001) 42–47.

[7] A. M. Pitangueira, R. S. P. Maciel, M. Barros, Software requirements selection and prioritization using sbse approaches: A systematic review and mapping of the literature, Journal of Systems and Software 103 (2015) 267–280.

[8] A. Ramírez, J. R. Romero, S. Ventura, An approach for the evolutionary discovery of software architectures, Information Sciences 305 (2015) 234–255.

[9] C. L. Simons, I. C. Parmee, Elegant object-oriented software design via interactive, evolutionary computation, Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 42 (6) (2012) 1797–1805.

[10] M. Harman, J. Clark, Metrics are fitness functions too, in: Software Metrics, 2004. Proceedings. 10th International Symposium on, IEEE, 2004, pp. 58–69.

[11] N. Fenton, J. Bieman, Software metrics: a rigorous and practical approach, CRC Press, 2014.

[12] M. O. Cinnéide, I. H. Moghadam, M. Harman, S. Counsell, L. Tratt, An experimental search-based approach to cohesion metric evaluation, Empirical Software Engineering (2016) 1–38.

[13] C. Simons, J. Singer, D. White, Software refactoring: Metrics are not enough, in: Proceedings of the 7th International Symposium on Search-Based Software Engineering (SSBSE 2015), Lecture Notes in Computer Science (LNCS) 9275, Springer, 2015, pp. 47–61.

[14] Z. Michalewicz, Genetic algorithms+ data structures= evolution programs, Springer Science & Business Media, 2013.

[15] D. Meignan, S. Knust, J.-M. Frayret, G. Pesant, N. Gaud, A review and taxonomy of interactive optimization methods in operations research, ACM Transactions on Interactive Intelligent Systems (TiiS) 5 (3) (2015) 17.

[16] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Transactions on Evolutionary Computation 6 (2) (2002) 182–197.

[17] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, M. Ó Cinnéide, High dimensional search-based software engineering: finding tradeoffs among 15

objectives for automating software refactoring using NSGA-III, in: Proceedings of the 2014 conference on Genetic and evolutionary computation (GECCO 2014), ACM, 2014, pp. 1263–1270.

[18] K. Deb, Multi-objective optimization using evolutionary algorithms, Vol. 16, John Wiley & Sons, 2001.

[19] H. Sackman, W. J. Erikson, E. E. Grant, Exploratory experimental studies comparing online and offline programming performance, Communications of the ACM 11 (1) (1968) 3–11.

[20] B. W. Boehm, The high cost of software, Practical Strategies for Developing Large Software Systems (1975) 3–15.

[21] G. R. Bergersen, D. I. Sjoberg, T. Dyba, Construction and validation of an instrument for measuring programming skill, IEEE Transactions on Software Engineering 40 (12) (2014) 1163–1184.

[22] Manifesto for agile software development, accessed 15 June 2016.
URL `http://www.agilemanifesto.org`

[23] C. L. Simons, Whither (away) software engineers in SBSE?, in: 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE), IEEE, 2013, pp. 49–50.

[24] G. Klein, D. D. Woods, J. M. Bradshaw, R. R. Hoffman, P. J. Feltovich, Ten challenges for making automation a team player, IEEE Intelligent Systems 19 (6) (2004) 91–95.

[25] P. Gregory, L. Barroca, H. Sharp, A. Deshpande, K. Taylor, The challenges that challenge: Engaging with agile practitioners concerns, Information and Software Technology.

[26] S. Freudenberg, H. Sharp, The top 10 burning research questions from practitioners, IEEE Software 27 (5) (2010) 8–9.

[27] H. Hemmati, L. Briand, A. Arcuri, S. Ali, An enhanced test case selection approach for model-based testing: an industrial case study, in: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, ACM, 2010, pp. 267–276.

[28] Y. Amannejad, V. Garousi, R. Irving, Z. Sahaf, A search-based approach for cost-effective software test automation decision support and an industrial case study, in: Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on, IEEE, 2014, pp. 302–311.

[29] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, M. Liaaen, Multi-objective test prioritization in software product line testing: an industrial case study, in: Proceedings of the 18th International Software Product Line Conference-Volume 1, ACM, 2014, pp. 32–41.

[30] C. L. Simons, J. Smith, P. White, Interactive ant colony optimization (iaco) for early lifecycle software design, Swarm Intelligence 8 (2) (2014) 139–157.

[31] B. Marculescu, R. Feldt, R. Torkar, S. Poulding, An initial industrial evaluation of interactive search-based testing for embedded software, Applied Soft Computing 29 (2015) 26–39.

[32] A. A. Araújo, M. Paixao, I. Yeltsin, A. Dantas, J. Souza, An architecture based on interactive optimization and machine learning applied to the next release problem, Automated Software Engineering (2016) 1–49.

[33] C. L. Simons, I. C. Parmee, Dynamic parameter control of interactive local search in UML software design, in: Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on, IEEE, 2010, pp. 3397–3404.

[34] A. Arcuri, G. Fraser, On parameter tuning in search based software engineering, in: International Symposium on Search Based Software Engineering, Springer, 2011, pp. 33–47.

[35] F. Wu, W. Weimer, M. Harman, Y. Jia, J. Krinke, Deep parameter optimisation, in: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, ACM, 2015, pp. 1375–1382.

[36] S. Yoo, M. Harman, S. Ur, GPGPU test suite minimisation: search based software engineering performance improvement using graphics cards, Empirical Software Engineering 18 (3) (2013) 550–593.

[37] W. B. Langdon, M. Harman, Optimising existing software with genetic programming, IEEE Transactions on Evolutionary Computation 19 (1) (2015) 118–135.

[38] D. Rodriguez, I. Herraiz, R. Harrison, On software engineering repositories and their open problems, in: Proceedings of the First International Workshop on Realizing AI Synergies in Software Engineering, IEEE Press, 2012, pp. 52–56.

[39] M. Harman, The role of artificial intelligence in software engineering, in: Proceedings of the First International Workshop on Realizing AI Synergies in Software Engineering, IEEE Press, 2012, pp. 1–6.

[40] L. Williams, What agile teams think of agile principles, Communications of the ACM 55 (4) (2012) 71–76.

[41] Scrum alliance, accessed 15 June 2016.
URL `http://www.scrumalliance.org`

[42] K. Schwaber, M. Beedle, Agile Software Development with Scrum, Prentice Hall, 2002.

[43] K. Beck, C. Andres, Extreme programming explained: embrace change, 2nd Edition, Addison-Wesley, 2005.

[44] A. Cockburn, Agile software development, Pearson Education, 2002.

[45] Atlassian JIRA, accessed 15 June 2016.
URL `https://www.atlassian.com/software/jira`

[46] E. Coelho, A. Basu, Effort estimation in agile software development using story points, International Journal of Applied Information Systems (IJAIS) 3 (7).

[47] I. M. Del Águila, J. Del Sagrado, Three steps multiobjective decision process for software release planning, Complexity.

[48] J. del Sagrado, I. M. del Águila, F. J. Orellana, Multi-objective ant colony optimization for requirements selection, Empirical Software Engineering 20 (3) (2015) 577–610.

[49] A. M. Pitangueira, P. Tonella, A. Susi, R. S. Maciel, M. Barros, Risk-aware multi-stakeholder next release planning using multi-objective optimization, in: International Working Conference on Requirements Engineering: Foundation for Software Quality, Springer, 2016, pp. 3–18.

[50] M. B. Chrissis, M. Konrad, S. Shrum, CMMI guidlines for process integration and product improvement, Addison-Wesley Longman Publishing Co., Inc., 2003.

[51] ISO IEC 15504-5:2012, information technology – process assessment – part 5: An exemplar software life cycle process assessment model, accessed 16 June 2016.
URL `http://www.iso.org/iso/catalogue_detail.htm?csnumber=60555`

[52] S. Lane, I. Richardson, Process models for service-based applications: A systematic literature review, Information and Software Technology 53 (5) (2011) 424–439.

[53] M. E. Nordberg III, Managing code ownership, IEEE Software 20 (2) (2003) 26–33.

[54] L. M. Maruping, X. Zhang, V. Venkatesh, Role of collective ownership and coding standards in coordinating expertise in software project teams, European Journal of Information Systems 18 (4) (2009) 355–371.

[55] J. L. Wilkerson, D. R. Tauritz, J. M. Bridges, Multi-objective coevolutionary automated software correction, in: Proceedings of the 14th annual conference on Genetic and Evolutionary Computation, ACM, 2012, pp. 1229–1236.

[56] B. R. Bruce, J. Petke, M. Harman, Reducing energy consumption using genetic improvement, in: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference, ACM, 2015, pp. 1327–1334.

[57] D. R. White, J. Singer, Rethinking genetic improvement programming, in: Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference, ACM, 2015, pp. 845–846.

[58] W. G. Griswold, W. F. Opdyke, The birth of refactoring: A retrospective on the nature of high-impact software engineering research, Software, IEEE 32 (6) (2015) 30–38.

[59] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley, 1999.

[60] M. Hafiz, J. Overbey, Refactoring myths, IEEE Software 32 (6) (2015) 39–43.

[61] B. Amal, M. Kessentini, S. Bechikh, J. Dea, L. B. Said, On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring, in: Search-Based Software Engineering, Springer, 2014, pp. 31–45.

[62] C. Simons, J. Smith, Exploiting antipheromone in ant colony optimisation for interactive search-based software design and refactoring, in: Proceedings of Conference Companion of Genetic and Evolutionary Computation Conference, ACM, 2016.

[63] W. B. Langdon, Evo_indent interactive evolution of gnu indent options., in: Proceedings of the Conference Companion of the Genetic and Evolutionary Computing Conference, 2009, pp. 2081–2084.

[64] M. Harman, Automated test data generation using search based software engineering, in: Automation of Software Test, 2007. AST'07. Second International Workshop on, IEEE, 2007, pp. 2–2.

[65] G. Fraser, M. Staats, P. McMinn, A. Arcuri, F. Padberg, Does automated unit test generation really help software testers? a controlled empirical study, ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (4) (2015) 23.